手抜きについて

手抜きチーム *

2023 年 3 月 28 日

本稿は第 33 回世界コンピュータ将棋選手権における手抜きのアピール文書です。

1　手抜きについて

手抜きは CSA プロトコルで対局を行うコンピュータ将棋プログラムです。開発者らが将棋のプログラムの仕組みを理解するために開発しています。
リポジトリ：`https://github.com/hikaen2/tenuki-d`

1.1　使用ライブラリ

『どうたぬき』(tanuki- 第 1 回世界将棋 AI 電竜戦バージョン) の評価関数ファイル nn.bin[*1]

1.2　特長

- NNUE
- α β 探索
- D 言語

1.3　Q&A

Q1. なぜ D 言語なのですか

A1. 1. 関数プロトタイプがいらない 2. 配列をスタック領域に置ける 3. ガベージコレクションがある 4. メモリ安全である 5. 速い からです

1. 関数プロトタイプがいらない

C/C++ では使っている関数のシグネチャをコンパイラに教えるために、プログラマが関数プロトタイプを書かなければなりません。これは DRY ではありません。関数のシグネチャは関数本体

* 鈴木太朗 @hikaen2, 玉川直樹 @Neakih_kick

*1 `https://github.com/nodchip/tanuki-/releases/tag/tanuki-denryu1`

に書いてあるのですから、本来はコンパイラが関数本体を見に行けばよいはずです。D（や Rust やその他の多くの言語）ではそのようになっているため、プログラマが関数プロトタイプを書く必要がありません。

2. 配列をスタック領域に置ける

Java や C#などの高級な言語では配列やオブジェクトはヒープ領域に置かれます。そのため、たとえば探索で配列やオブジェクトを作るとそのたびにヒープアロケーションが発生して時間がかかります。配列をスタック領域に置くのであれば、スタックポインタを進めるだけなのでとくに時間がかかりません。D では配列やオブジェクトをスタック領域に置くことができます。

3. ガベージコレクションがある

普通のプログラム（システムプログラムでないプログラム）を書くのであればガベージコレクションがあったほうが便利だと思います。C や C++ や Rust はシステムプログラミング言語なのでガベージコレクションがありません。D はガベージコレクションがあります。

4. メモリ安全である

C/C++ は配列の境界チェックをしないためメモリ安全ではありません。C/C++ で配列の境界を超えたアクセスは未定義動作であり、わかりにくく再現性のないバグを引き起こします。D は配列の境界チェックをチェックをするのでメモリ安全です。

5. 速い

ldc2[*2]でコンパイルした D のコードは、clang でコンパイルした C のコードと同等の速さが期待できると思います。

Q2. Rust でもよいのではありませんか

A2. Rust でもよいと思いますが、普通のプログラムを書くのであればガベージコレクションがあったほうが便利だと思います。

2　付録 1：私のための Minimax 入門

本章では私にむけて Minimax の解説をします。

関連する前提知識：再帰関数、木構造、ゲーム木、深さ優先探索、Ruby など

また、次の内容は本解説の対象外です：合法手生成、静止探索、静的評価、反復深化、置換表 など

2.1　Minimax

2.1.1　やりたいこと

図 1 のゲーム木で局面 a の評価値を求めたい。どうすればよいか。ただし先手番の局面が△、後手番の局面が▽とする。△、▽の中に先手から見た評価値を記入するものとする。

[*2] `https://github.com/ldc-developers/ldc`

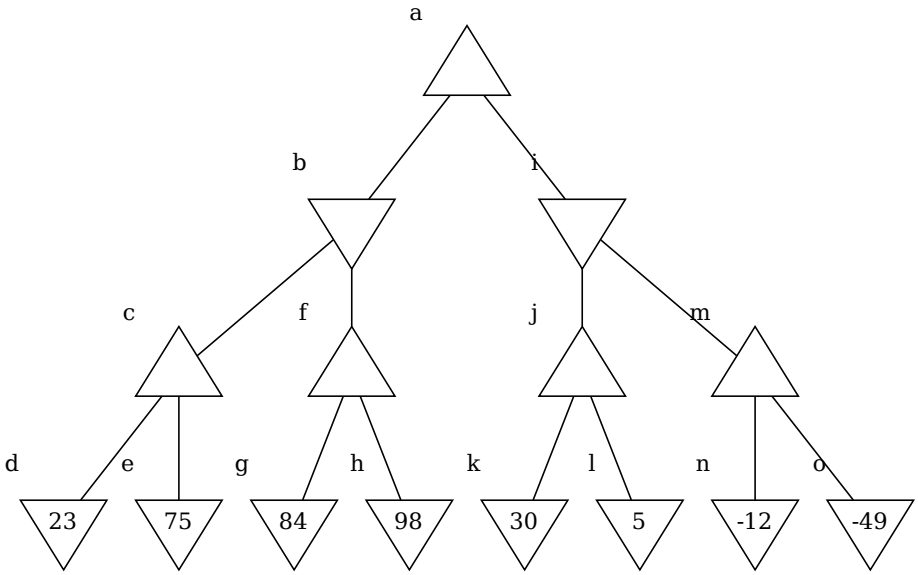


図 1

2.1.2 手順

まず c と f の評価値を求める。c と f は先手番なので子局面の中からなるべく評価値の大きいものを選ぶ。すなわち：

$c = \max(d, e) = \max(23, 75) = 75$

$f = \max(g, h) = \max(84, 98) = 98$

つぎに b の評価値を求める。b は後手番なので子局面の中からなるべく評価値の小さいものを選ぶ。すなわち：

$b = \min(c, f) = \min(75, 98) = 75$

同じように j と m と i の評価値を求める：

$j = \max(k, l) = \max(30, 5) = 30$

$m = \max(n, o) = \max(-12, -49) = -12$

$i = \min(j, m) = \min(30, -12) = -12$

最後に a の評価値を求める：

$a = \max(b, i) = \max(75, -12) = 75$

以上の結果を記入すると図 2 になる。

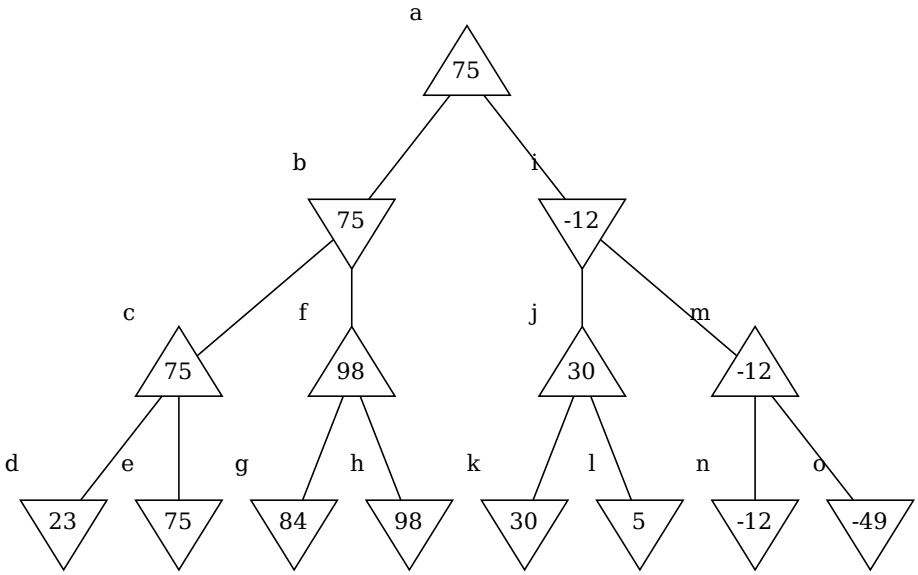


図 2

また、以上の結果を一つの式にまとめると：

$$
\begin{aligned}
a &= \max\Big(\min\big(\max(d,e),\max(g,h)\big),\min\big(\max(k,l),\max(n,o)\big)\Big) \\
&= \max\Big(\min\big(\max(23,75),\max(84,98)\big),\min\big(\max(30,5),\max(-12,-49)\big)\Big) \\
&= \max\Big(\min\big(75,98\big),\min\big(30,-12\big)\Big) \\
&= \max\Big(75,-12\Big) \\
&= 75
\end{aligned}
\tag{1}
$$

2.1.3 コード (Ruby)

リスト 1: minimax.rb

```
tree =
  {side: :MAX, children: [
    {side: :MIN, children: [
      {side: :MAX, children: [
        {side: :MIN, value: 23},
        {side: :MIN, value: 75},
      ]},
      {side: :MAX, children: [
        {side: :MIN, value: 84},
        {side: :MIN, value: 98},
      ]},
    ]},
    {side: :MIN, children: [
```

```
14          {side: :MAX, children: [
15            {side: :MIN, value: 30},
16            {side: :MIN, value: 5},
17          ]},
18          {side: :MAX, children: [
19            {side: :MIN, value: -12},
20            {side: :MIN, value: -49},
21          ]},
22        ]},
23      ]}
24
25  def minimax(p)
26    return p[:value] if p[:children] == nil
27    return p[:children].map{|q| minimax(q)}.max if p[:side] == :MAX
28    return p[:children].map{|q| minimax(q)}.min if p[:side] == :MIN
29  end
30
31  puts minimax(tree) # => 75
```

1 行目で定義している tree が図 1 のゲーム木である。

25 行目で minimax 関数を定義している。p は Position（局面）の頭文字である。q は p の子局面である。

26 行目は終端局面のときのコードである。（静的）評価値を返している。

27 行目は先手の場合のコードである。子局面の minimax 値のうち最も大きいものを返している。

28 行目は後手の場合のコードである。子局面の minimax 値のうち最も小さいものを返している。

2.1.4 Q&A

Q1. なぜ Ruby なのですか

A1. 簡潔で表現力が高いので説明に適していると考えたからです。疑似コードと違い実際に実行できるという利点もあります。一方、C や D や Rust に比べて実行速度は遅いので、実戦には向きません。

Q2. 式 (1) からどのようにしたらリスト 1 が得られるのですか

A2. 式 (1) は min と max が再帰的に連なっています。そこでどうにかして式 (1) を次の漸化式にします：

$$\mathrm{minimax}(p) = \begin{cases} \text{value of } p & (p\text{が終端局面のとき}) \\ \max\limits_{q \in \text{children of } p} \mathrm{minimax}(q) & (\text{上記以外で}p\text{が先手番のとき}) \\ \min\limits_{q \in \text{children of } p} \mathrm{minimax}(q) & (\text{上記以外で}p\text{が後手番のとき}) \end{cases} \tag{2}$$

これを Ruby で書くとリスト 1 が得られます。

2.2 Negamax

リスト 1 では先手か後手かで場合分けしている（27 行目と 28 行目）。Negamax ではこれを先後の区別なく共通化する。子局面の評価値の最小を求めることは、子局面の評価値の正負を反転して最大を求め、結果の正負を反転することと同じである。すなわち $\min(x, y) = -\max(-x, -y)$。これを用いて式 (1) を変形すると：

$$
\begin{aligned}
a &= \max\Big(\min\big(\max(d,e),\max(g,h)\big),\min\big(\max(k,l),\max(n,o)\big)\Big) \\
&= \max\Big(-\max\big(-\max(d,e),-\max(g,h)\big),-\max\big(-\max(k,l),-\max(n,o)\big)\Big) \\
&= \max\Big(-\max\big(-\max(23,75),-\max(84,98)\big),-\max\big(-\max(30,5),-\max(-12,-49)\big)\Big) \\
&= \max\Big(-\max\big(-75,-98\big),-\max\big(-30,-(-12)\big)\Big) \\
&= \max\Big(-(-75),-12\Big) \\
&= 75
\end{aligned}
\tag{3}
$$

2.2.1 コード (Ruby)

tree の定義はリスト 1 と同じなので省略する。以下同様。

リスト 2: negamax.rb

```
1  def negamax(p)
2    return p[:side] == :MAX ? p[:value] : -p[:value] if p[:children] == nil
3    p[:children].map{|q| -negamax(q)}.max
4  end
5
6  puts negamax(tree) # => 75
```

Minimax が常に先手から見た評価値を返すのに対して、Negamax は手番がある側から見た評価値を返す（表 1）。たとえば後手番の局面で Negamax が正の値が返したら、それは後手が有利（先手が不利）という意味である。そのために 2 行目では、後手番であれば評価値の符号を反転して返している。言い換えると、Negamax では静的評価関数が手番のある側から見た評価値を返す必要がある。

3 行目には -negamax(q) という式がある。マイナスが付いているのは相手の言い分を反転させるためである。たとえば後手が「わたしは +100 点である」と言うのなら、それは先手にとっては −100 点の意味だからである。

2.2.2 Q&A

Q1. Minimax を Negamax にするとどのくらいよいですか

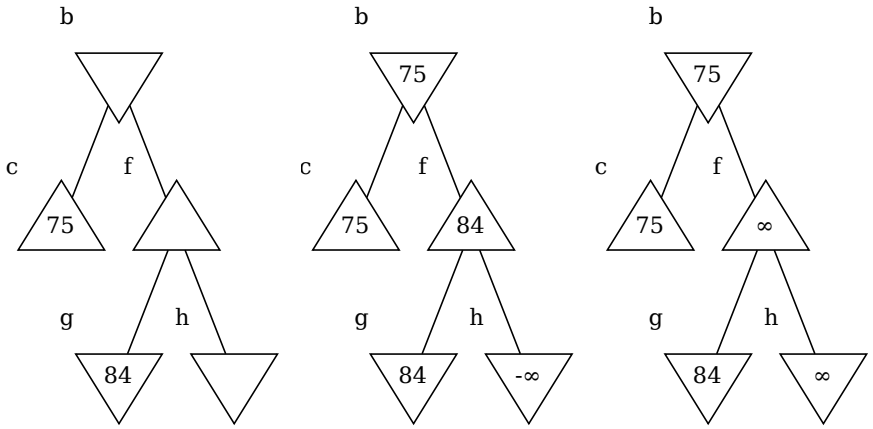
表 1: Minimax 値/Negamax 値の正負

局面	Minimax 値	Negamax 値
先手有利 かつ 先手番	正	正
先手有利 かつ 後手番	正	負
後手有利 かつ 先手番	負	負
後手有利 かつ 後手番	負	正

A1. 性能は変わりません。先手と後手の場合分けが一つなくなるのでコードが簡潔になります。これは後述する Alpha-Beta Pruning を適用したときにとくにうれしいです。

2.3 Alpha-Beta Pruning

Minimax では局面 a の値を求めるために子局面の全ての値を計算したが、実際のところ全ての値を計算する必要はない。たとえば図 3a の h はどのような値であっても b に影響を及ぼすことがない。仮に h に $-\infty$ と ∞ を入れた結果を図 3b と図 3c に示す。どちらも b は 75 であり、h の影響がない。このことから h は探索しなくてよいことがわかる。

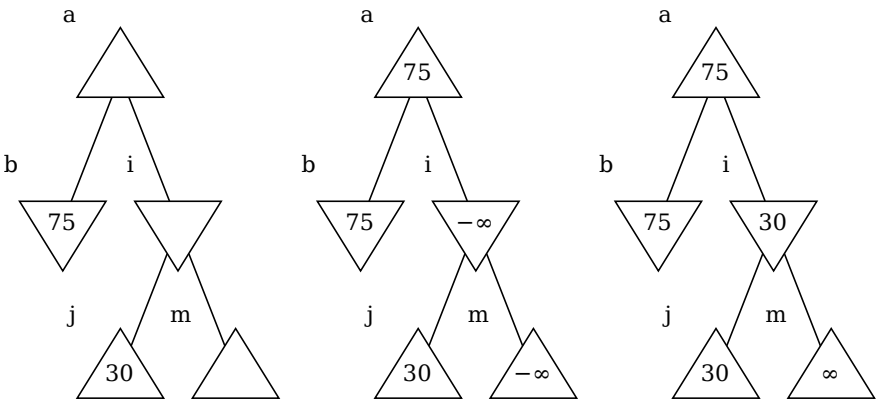


(a) 評価中の b の周辺　(b) h に $-\infty$ を入れた　(c) h に ∞ を入れた

図 3

なぜこのようになっているのかを説明する。① $c = 75$ で、なおかつ b は後手番なので $b = \min(75, f)$ であり、f が 75 を下回らない限り b に採用されないことがわかる。②一方 $g = 84$ で、なおかつ f は先手番なので $f = \max(84, h)$ であり、h がどのような値であっても f は 84 を下回らないことがわかる。①、②より f は h の値に関わらず b に採用されないことがわかる。したがって h を探索する必要がない。

同様に図 4 では局面 m の値を計算する必要がない。



(a) 評価中の a の周辺　(b) m に $-\infty$ を入れた　(c) m に ∞ を入れた

図 4

2.3.1 コード (Ruby)

リスト 3: alphabeta.rb

```
1  def alphabeta(p, alpha, beta)
2    raise 'assertion error' unless alpha < beta
3    return p[:value] if p[:children] == nil
4    if p[:side] == :MAX
5      p[:children].each do |e|
6        alpha = [alpha, alphabeta(e, alpha, beta)].max
7        return beta if beta <= alpha
8      end
9      return alpha
10   else
11     p[:children].each do |e|
12       beta = [beta, alphabeta(e, alpha, beta)].min
13       return alpha if beta <= alpha
14     end
15     return beta
16   end
17 end
18
19 puts alphabeta(tree, -Float::INFINITY, Float::INFINITY) # => 75
```

alpha と beta は探索する評価値の範囲を表す窓である。alpha が窓の下限、beta が窓の上限であり、alpha < beta である必要がある（2 行目）。初期値は $-\infty$ 〜 $+\infty$ である（19 行目）。窓は開区間である。すなわち alpha $< x <$ beta の x が窓に収まる値である。

探索中に評価値が窓に収まらないことがわかった場合は、その局面は実現しないため探索を打ち切ることができる。探索を打ち切る場合の評価値は、alpha 以下の適当な値や、beta 以上の適当な値を返しておけばよい。このとき、ちょうど alpha や beta を返す方法を fail-hard という。alpha 以下や beta 以上の、より真の値に近い値を返す方法を fail-soft という。

探索中に先手番は alpha を押し上げながら探索を進める（6 行目）。このとき、alpha が beta 以上になれば、その局面は窓の範囲外なので beta を返して終了する（7 行目）。これを beta-cutoff という。

探索中に後手番は beta を引き下げなから探索を進める（12 行目）。このとき、beta が alpha 以下になれば、その局面は窓の範囲外なので alpha を返して終了する（13 行目）。これを alpha-cutoff という。

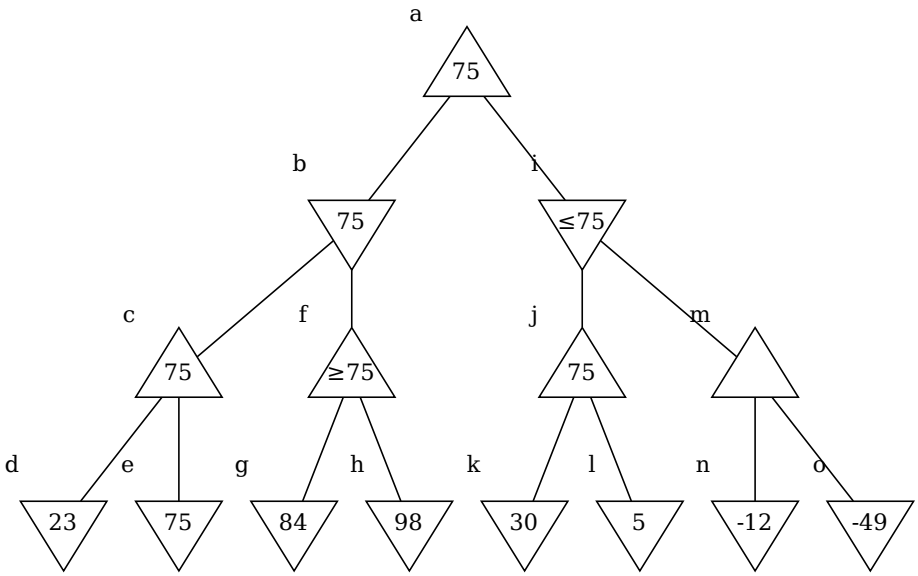


図 5

2.3.2 Q&A

Q1. Minimax を Alpha-Beta Pruning するとどのくらいよいですか

A1. ゲーム木の 1 局面あたりの枝の数を d (degree)、ゲーム木の高さを h (height) とすると、Minimax の計算量が $O(d^h)$、Alpha-Beta Pruning の計算量が最良で $O(d^{h/2})$ なのでだいぶよいです [1]。

Q2. 7 行目の beta <= alpha を beta < alpha にしたらどうなりますか

A2. これは alpha が窓の上限を越えたかどうかを判定する式です。窓は開区間なので alpha == beta のときにすでに alpha は窓の上限を越えています。したがって判定式を beta < alpha にすると、ちょうど alpha == beta のときに alpha が本当は上限を越えているのに上限を越えていないと判定されることになり、枝刈りが弱くなります。

Q3. 7 行目の return beta を return alpha にしたらどうなりますか

A3. beta-cutoff を fail-soft にするということになります。ここでは beta 以上の値を返せばよく、このときの alpha は beta 以上なので alpha を返しても問題ありません。ただし評価値が alpha より下だったときに 9 行目で fail-hard しています。したがって、評価値が beta より上のときは fail-soft して、評価値が alpha より下のときは fail-hard するという一貫性のないコードになってしまいます。全体を fail-soft で実装する方法については 2.5 節で説明します。

2.4 Negamax Alpha-Beta Pruning

Negamax を Alpha-Beta Pruning することができる。Negamax Alpha-Beta Pruning はただの Negamax と同様に、手番のある側から見た評価値を返す。

2.4.1 コード (Ruby)

リスト 4: nega-alphabeta.rb

```
1  def nega_alphabeta(p, alpha, beta)
2    raise 'assertion error' unless alpha < beta
3    return p[:side] == :MAX ? p[:value] : -p[:value] if p[:children] == nil
4    p[:children].each do |e|
5      alpha = [alpha, -nega_alphabeta(e, -beta, -alpha)].max
6      return beta if beta <= alpha
7    end
8    alpha
9  end
10
11 puts negaalpha(tree, -Float::INFINITY, Float::INFINITY) # => 75
```

5 行目の -beta, -alpha では窓をひっくり返している。たとえば窓が $-100 \sim +200$ であれば、ひっくり返した窓は $-200 \sim +100$ となる。これはたとえば先手が持つ $-100 \sim +200$ の窓は、後手から見ると $-200 \sim +100$ の窓だからである。

2.4.2 Q&A

Q1. Minimax Alpha-Beta Pruning と Negamax Alpha-Beta Pruning はどちらがいいですか

A1. 性能は同じです。Negamax Alpha-Beta Pruning のほうがコードが簡潔なのがいいです。

2.5 fail-soft Alpha-Beta

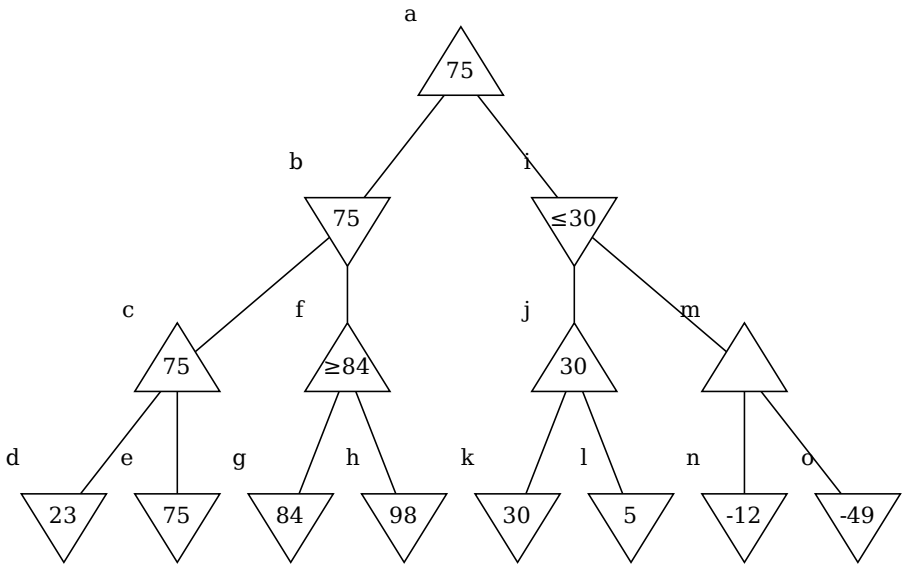


図 6

2.5.1 コード (Ruby)

リスト 5: failsoft-alphabeta.rb

```ruby
def failsoft_alphabeta(p, alpha, beta)
  raise 'assertion error' unless alpha < beta
  return p[:side] == :MAX ? p[:value] : -p[:value] if p[:children] == nil
  score = -Float::INFINITY
  p[:children].each do |q|
    score = [score, -failsoft_alphabeta(q, -beta, -alpha)].max
    alpha = [alpha, score].max
    return score if beta <= alpha
  end
  score
end

puts alphabeta_failsoft(tree, -Float::INFINITY, Float::INFINITY) # => 75
```

2.5.2 Q&A

Q1. fail-hard と fail-soft はどっちがいいですか

A1. 性能は同じです。コードは fail-hard のほうがちょっと簡単です。本稿では扱いませんが置

換表に評価値を保存している場合は fail-soft のほうが真の値に近いのでよさそうな気がします [2]。

2.6 Principal Variation Search

2.6.1 コード (Ruby)

リスト 6: pv-search.rb

```ruby
def pv_search(node, alpha, beta)
  return p[:side] == :MAX ? p[:value] : -p[:value] if p[:children] == nil
  score = -Float::INFINITY
  first = true
  p[:children].each do |q|
    if first
      v = -pv_search(q, -beta, -alpha)
      first = false
    else
      v = -pv_search(q, -alpha - 1, -alpha)
      v = -pv_search(q, -beta, -alpha) if alpha < v && v < beta
    end
    score = [score, v].max
    alpha = [alpha, score].max
    return score if beta <= alpha
  end
  score
end
```

2.6.2 Q&A

Q1. 8 行目を first = false if alpha < v としている実装がありますが、どっちがいいですか

A1. わかりません

3 付録 2：関数についての考察

高階関数は、関数を受け取る関数や、関数を返す関数と説明される。しかしこの説明には関数とは何であるかの説明が足りていない。そこで本章では関数について考察する。

歴史的経緯により関数には次の 2 通りの定義がある [3]：

1. 古典的定義：相伴って変化する数（あるいは量）；すなわち関数は数である
2. 近代的定義：（一意）対応、あるいは対応関係；すなわち関数は対応関係である

ここで数と対応関係は明らかに異なるものであることに注意されたい。たとえば 1 や 2 や 3 が数である。一方、対応関係とは x に対して x^2 が対応するといった関係である。

古典的には関数はたとえば $y = x^2$ とか $f(x) = x^2$ のように書かれ、それぞれ「y は x の関数で

ある (y is a function of x)」、「$f(x)$ は x の関数である ($f(x)$ is a function of x)」、あるいは単に「y は関数である (y is a function)」、「$f(x)$ は関数である ($f(x)$ is a function)」と呼ばれる。

ここで y や $f(x)$ は、x に相伴って変化する数である。したがって関数は数である。ここでは f は関数を表すためのただの記号である。

一方、近代的定義では関数は数ではなく、対応関係であると解釈される。そこでは f こそが関数であり、$f(x)$ は「関数 f の x における値 (value of a function f at x)」と呼ばれる。したがって関数は対応関係である。

以上の違いを表 2 に記す。

表 2

	古典的定義では	近代的定義では
f は	関数を表す記号	関数（すなわち対応関係）
$f(x)$ は	x の関数（すなわち数）	関数 f の x における値

古典的定義においては f はただの記号だが、近代的定義においては f こそが関数である。そこで近代的定義においては f を数式で表す。たとえば式 $f(x) = x^2$ において、f は矢印表記を用いると $f : x \mapsto x^2$ と表せる。ラムダ式を用いると $f = \lambda x.x^2$ と表せる。

関数に近代的定義を導入すると、関数の対応関係そのものを論ずることができる。冒頭で、高階関数とは、関数を受け取る関数や、関数を返す関数であると述べた。ここで関数とは対応関係のことである。すなわち高階関数とは、対応関係を受け取る関数や、対応関係を返す関数と言うことができる。これは高階でない関数が、数を受け取り、数を返すのと比べて、明らかに異なる。

たとえば、受け取った関数を 2 回適用する関数を返す高階関数 apply_twice をラムダ式を用いて次のように定義できる：apply_twice $= \lambda f.\lambda x.f(f(x))$

ここで $\lambda f.\lambda x.f(f(x))$ は f を受け取り (λf)、$\lambda x.f(f(x))$ を返す関数を意味する。返す値である $\lambda x.f(f(x))$ もまた関数である。したがって $\lambda f.\lambda x.f(fx)$ は関数を返す関数であり、高階関数である。

参考文献

[1] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, Vol. 6, pp. 293–326, 1975.

[2] John P. Fishburn. Another optimization of alpha-beta search. *SIGART Newsl.*, Vol. 84, pp. 37–38, 1983.

[3] 公田藏. 近代日本における, 函数の概念とそれに関連したことがらの受容と普及. 数理解析研究所講究録, Vol. 1787, pp. 265–279, 2012.