2013年6月7日

発見的探索

西田豊明

状態空間探索

状態空間探索問題(state space search problem)とは,

・状態集合　S

・初期状態　$s_0 \in S$

・状態に適用される述語　$P : S \to \{\mathrm{T}, \mathrm{F}\}$

・基本操作の集合　$O : \{o_i \mid o_i(s_1) = s_2\}$　　$s_1, s_2 \in S$

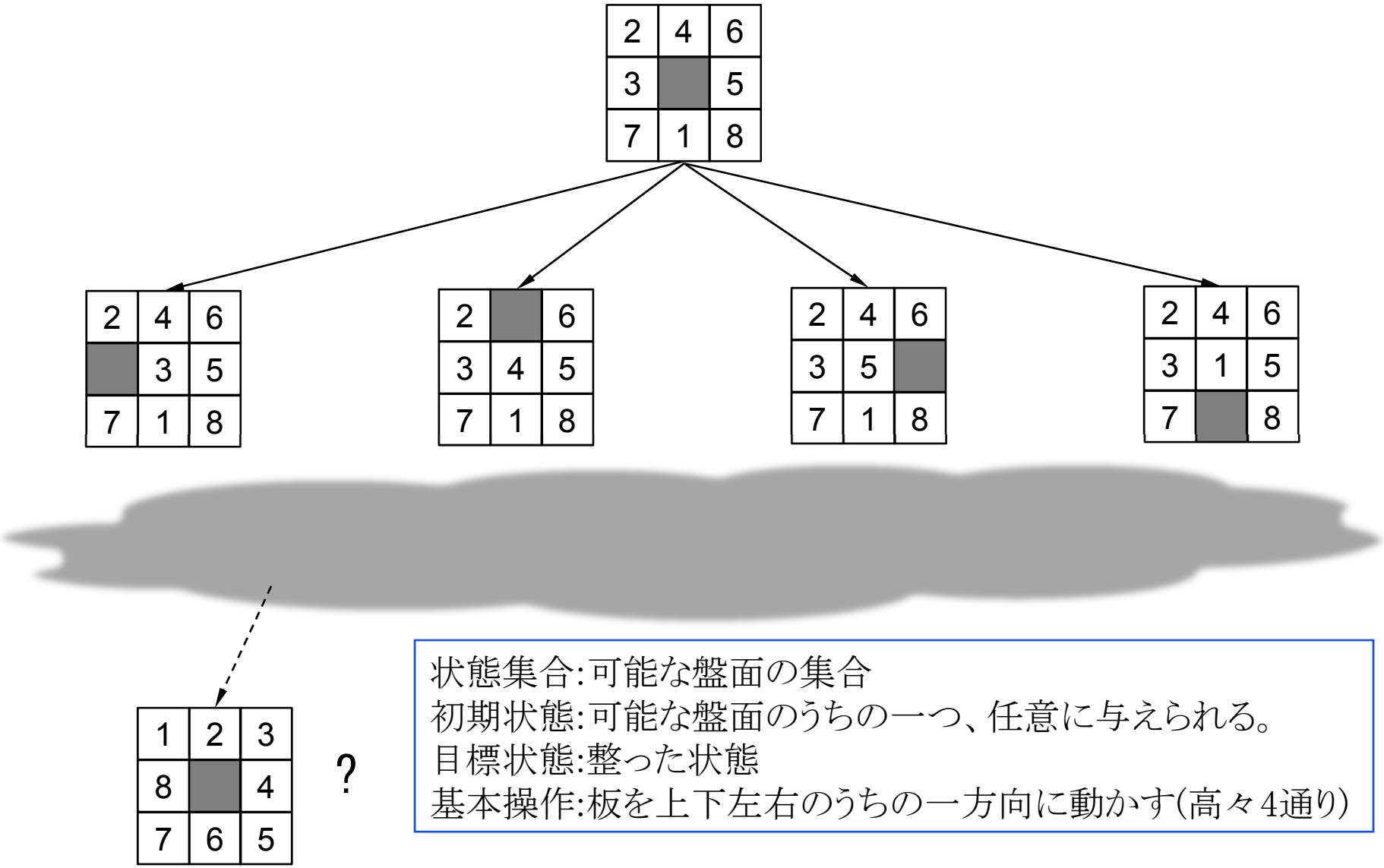
が与えられたとき,

$$P(o_{j_n}(\cdots o_{j_1}(s_0)\cdots)) = \mathrm{T}$$

となる操作の系列　$o_{j_1}, \cdots, o_{j_n}$ を求めよ, という問題である.

[西田 1999]

状態空間探索と8パズル



状態集合:可能な盤面の集合
初期状態:可能な盤面のうちの一つ、任意に与えられる。
目標状態:整った状態
基本操作:板を上下左右のうちの一方向に動かす(高々4通り)

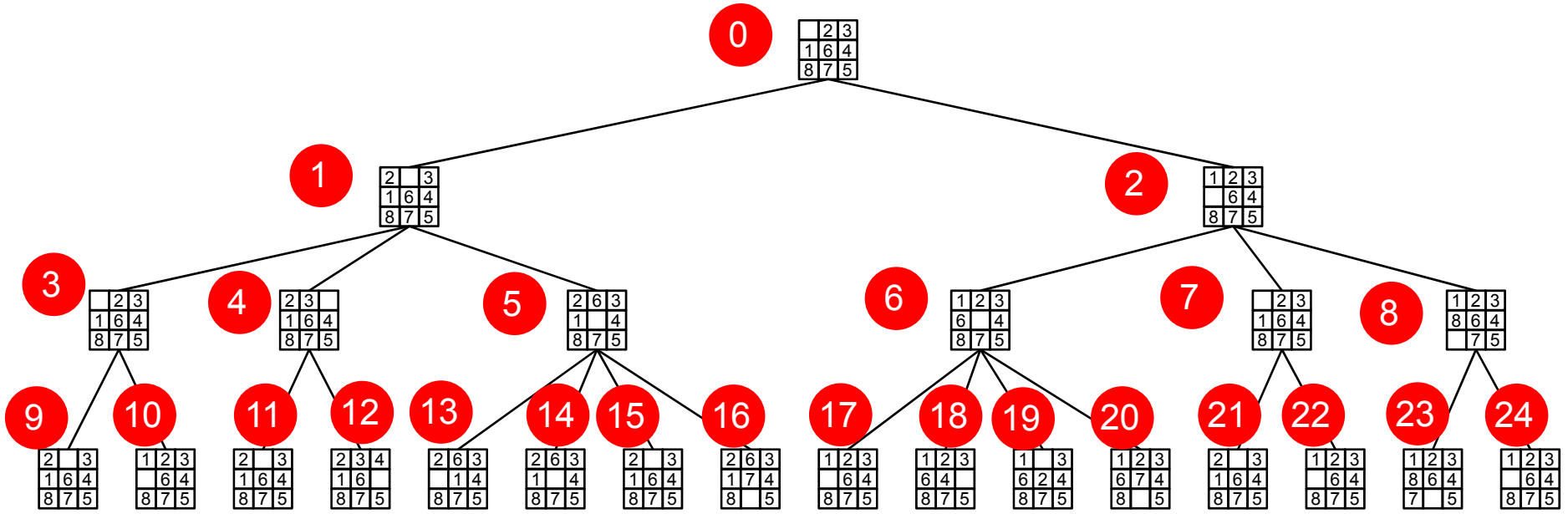
[西田 1999]

基本的な探索アルゴリズム

横型探索 (breadth-first search)

```
(setq open-list (initialize-open-list))
(loop
   (if (null open-list) (return nil))
   (setq node (pop open-list))
   (if (p node) (return node))
   (setq x (expand node))
   (setq open-list (append open-list x)))
```

青字のところは個別に定義する



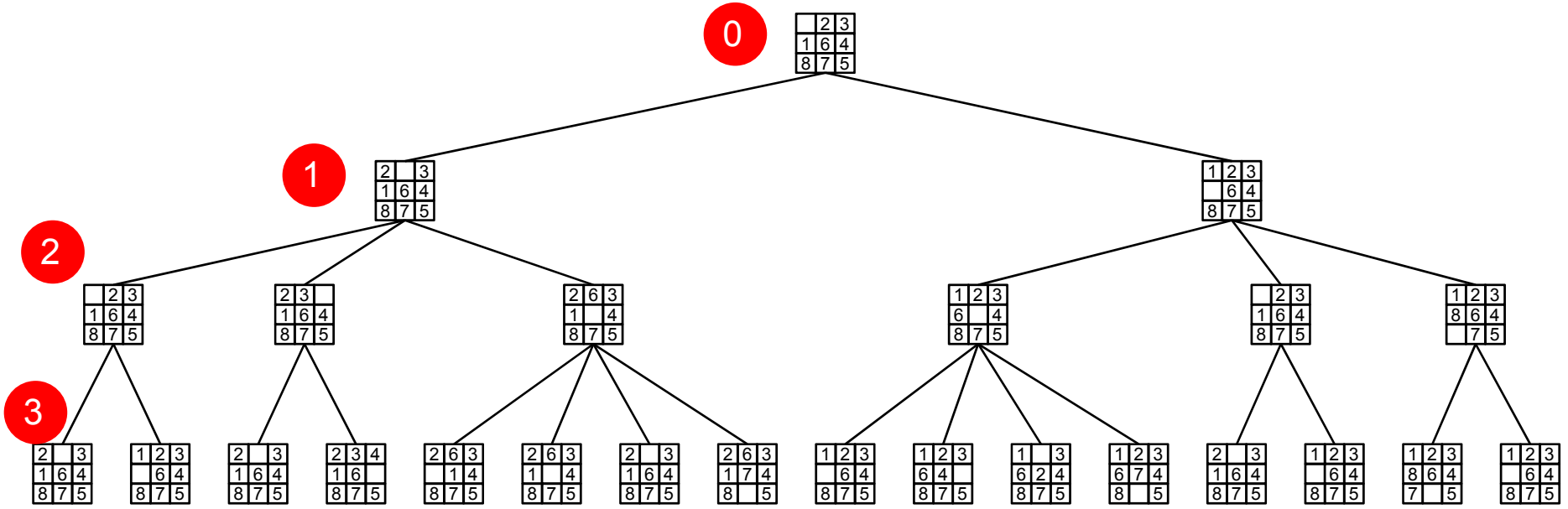
[西田 1999]

基本的な探索アルゴリズム

縦型探索 (depth-first search)

```
(setq open-list (initialize-open-list))
(loop
   (if (null open-list) (return nil))
   (setq node (pop open-list))
   (if (p node) (return node))
   (setq x (expand node))
   (setq open-list (append x open-list)))
```

青字のところは個別に定義する



[西田 1999]

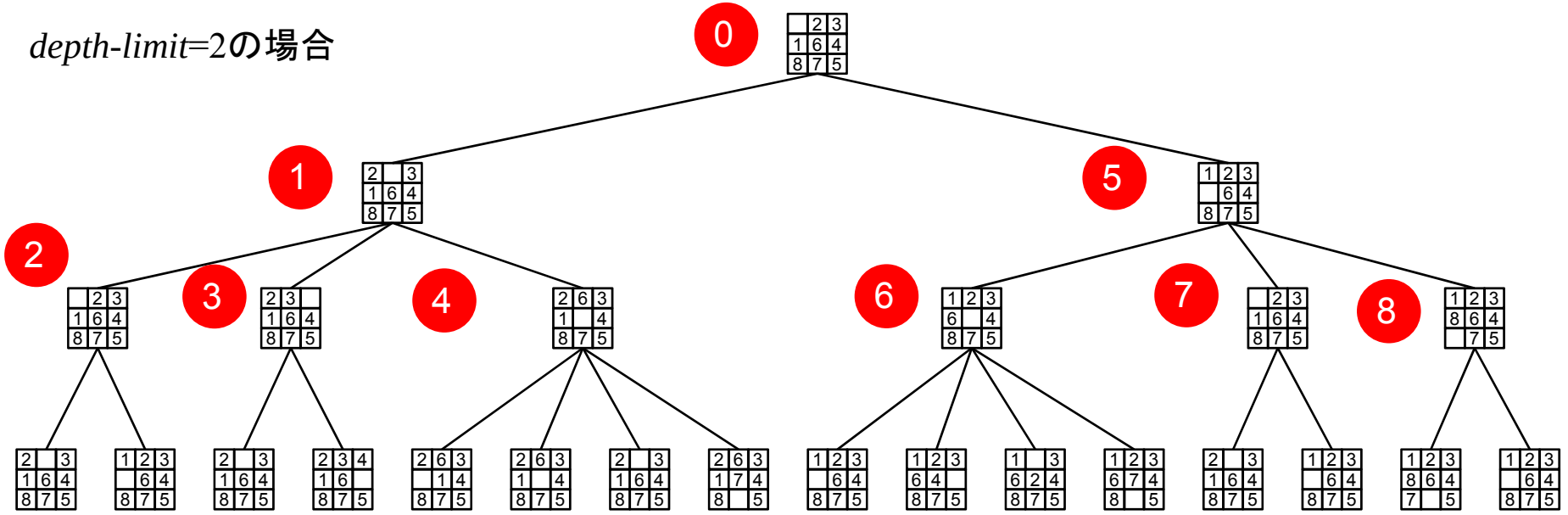
基本的な探索アルゴリズム

深さ制限縦型探索 (depth-limited search)

```
(setq open-list-2 (mapcar #'(lambda (x) `(,0 ,x)) open-list))
(loop
   (if (null open-list-2) (return nil))
   (setq pair (pop open-list-2))
   (setq d (first pair))
   (setq node (second pair))
   (if (p node) (return node))
      (cond ((< d depth-limit)
                (setq expanded (expand node))
                (setq a (mapcar #'(lambda (x) `(,(1+ d) ,x)) expanded))
                (setq open-list-2 (append a open-list-2)))))
```

青字のところは個別に定義する

*depth-limit=2*の場合

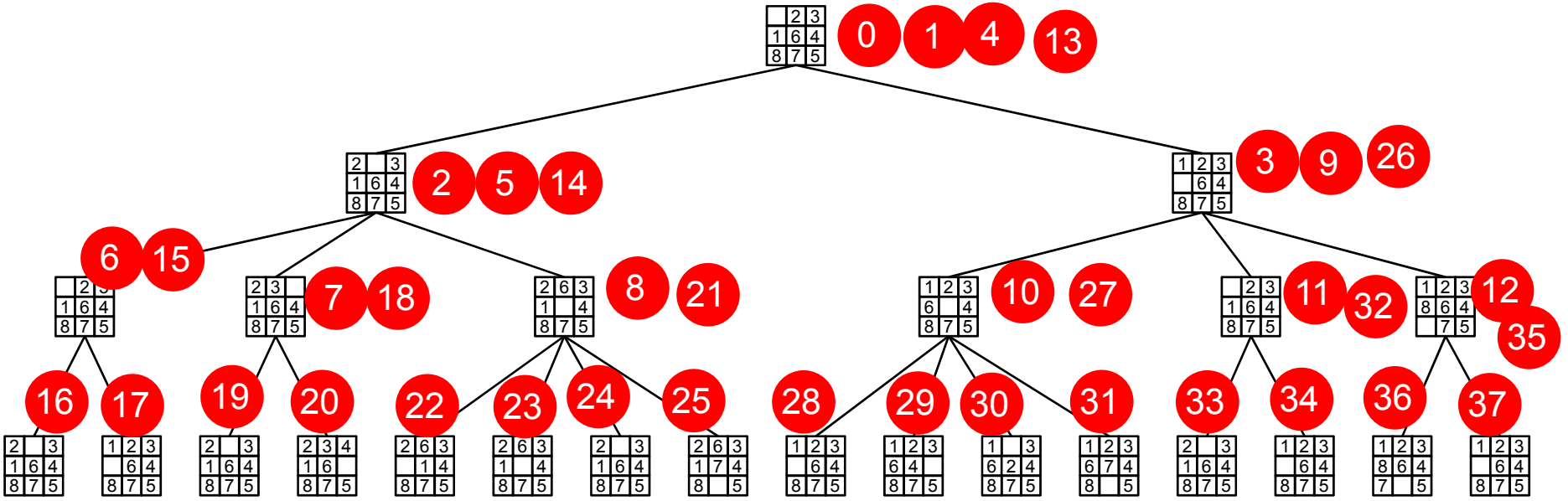


[西田 1999]

基本的な探索アルゴリズム

反復深化探索 (iterative deepening search)

```
(setq depth 0)
(loop
  (setq result (depth-limited-search depth …))
  (if result (return result))
  (setq depth (1+ depth)))
```



[西田 1999]

ヒューリスティックを用いた状態空間探索

状態空間探索では，コスト関数が与えられ，初期状態から最小コストで解に至る操作系列を見つけることが求められることが多い．

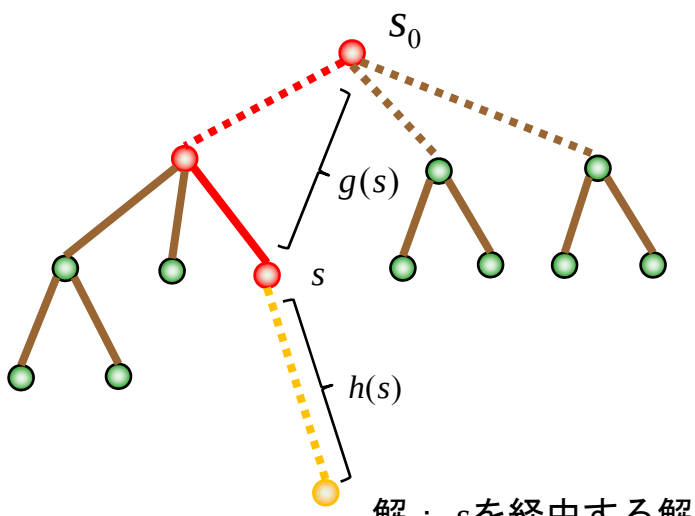
操作系列 $p = \{o_1, \cdots, o_n\}$ のコスト $c(p)$ が

$$c(p) = \sum_i c(o_i)$$ … p を構成する各操作 c_i のコスト $p(c_i)$ の和

で与えられる場合，状態 s を経由して最小コストで解に至る操作系列のコストを $f(s)$ とすると，

$$f(s) = g(s) + h(s)$$

である．



どのノードを経由すれば最小コストで解に至るかわからないし，
我々が興味を持つ問題については，
探索過程では，$h(s)$ はわからない
（わかるのは簡単すぎる問題）．

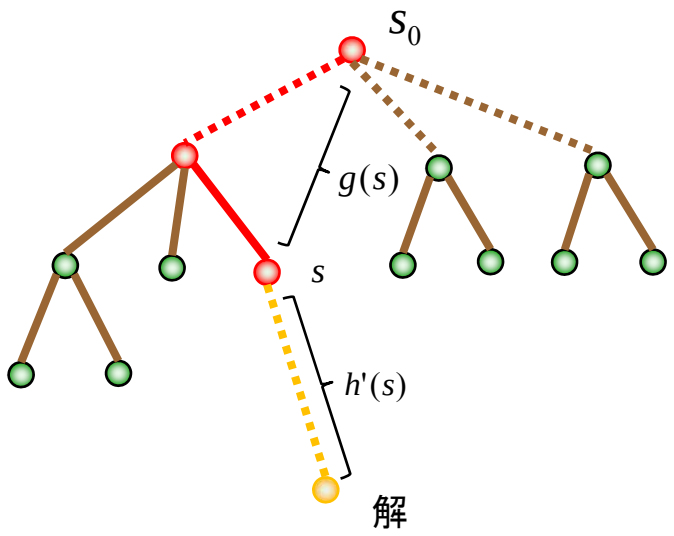
$h(s)$ の代わりに推定値 $h'(s)$ を用いる（h'はヒューリスティック関数と呼ばれる）．

解： sを経由する解のなかでコストが最小となるもの

[西田 1999]

ヒューリスティック探索の代表的方法

- 山登り法：$h'(s)$しか考慮しない．局所的な最適性を追求する．
- 最良優先探索：$h'(s)$しか考慮しない．大域的な最適性を追求する．
- A*アルゴリズム：$g(s)+h'(s)$を考慮する．大域的な最適性を追求する．$h'(s)$が楽天的ヒューリスティック（常に，$h'(s) \leq h(s)$ ）ならば，最初に見つける解の最適性が保証される．

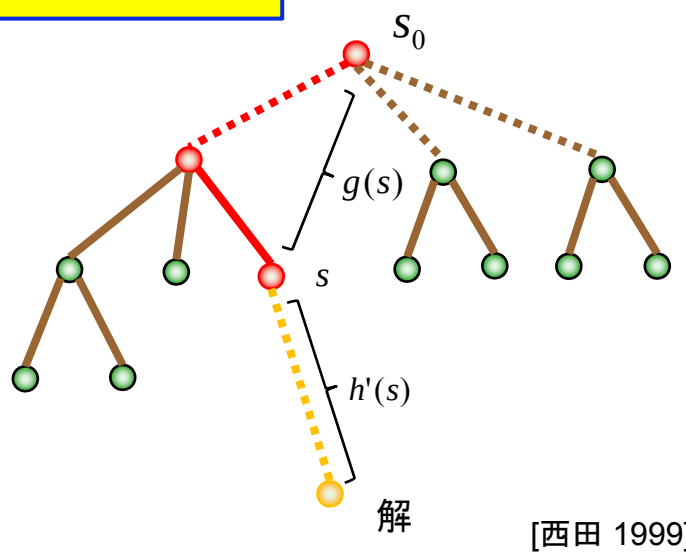


[西田 1999]

山登り法

```
(setq open-list (mapcar #'(lambda (x) `(0 0 ,x)) open-list-3))
(loop
   (if (null open-list-3) (return nil))
   (setq item (pop open-list-3))
   (setq state (third item))
   (if (p state) (return state))
   (setq d0 (state-expander state))
   (setq d (mapcar
               #'(lambda (x)
                    `(,(h- (second x))
                     ,(+ g (first x))
                     ,(second x)))
               d0))
   (setq e (add-to-open-list-for-a-star d nil))
   (setq open-list-3 (append e open-list-3)))
```

青字のところは個別に定義する

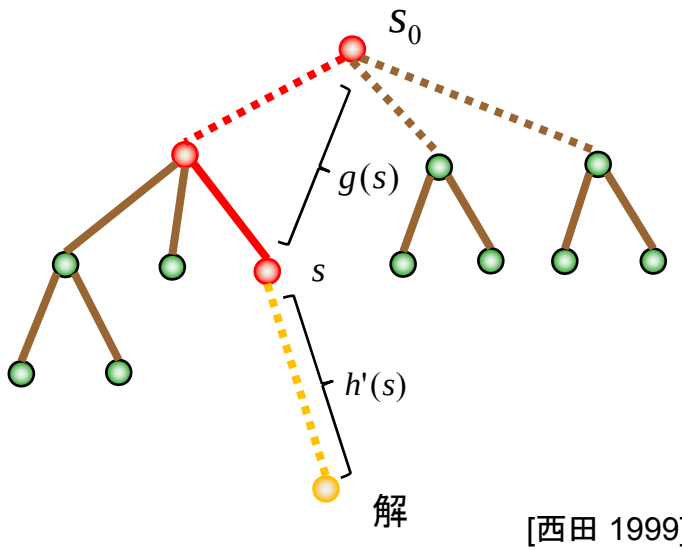


[西田 1999]

最良優先探索

```
(setq open-list-3 (mapcar #'(lambda (x) `(0 0 ,x)) open-list))
(loop
   (if (null open-list-3) (return nil))
   (setq item (pop open-list-3))
   (setq state (third item))
   (if (p state) (return state))
   (setq d0 (expand state))
   (setq d (mapcar
             #'(lambda (x) `(,(h- (second x))
                             ,(+ g (first x))
                             ,(second x))) d0))
   (setq open-list-3 (add-to-open-list-for-a-star d open-list-3)))
```

青字は関数引数

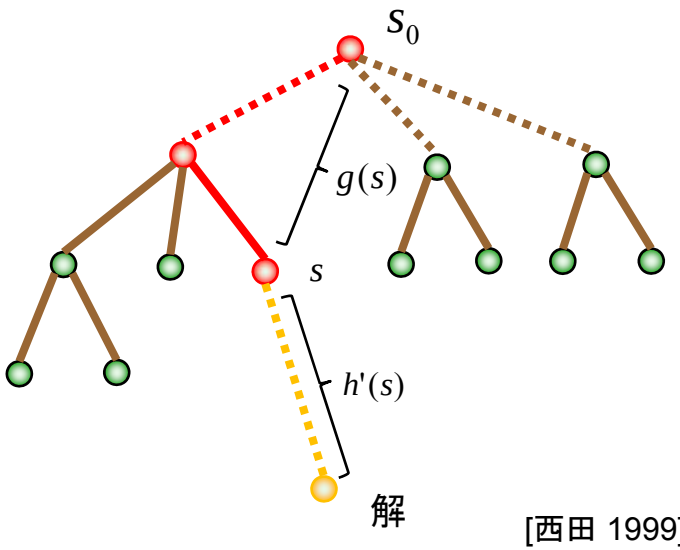


[西田 1999]

A*アルゴリズム

```
(setq open-list-3 (mapcar #'(lambda (x) `(0 0 ,x)) open-list))
(loop
   (if (null open-list-3) (return nil))
   (setq item (pop open-list-3))
   (setq g (second item))
   (setq state (third item))
   (if (p state) (return state))
   (setq d0 (expand state))
   (setq d (mapcar
             #'(lambda (x)
                 `(,(+ g (h- (second x)))
                   ,(+ g (first x))
                   ,(second x))) d0))
   (setq open-list-3 (add-to-open-list-for-a-star d open-list-3)))
```

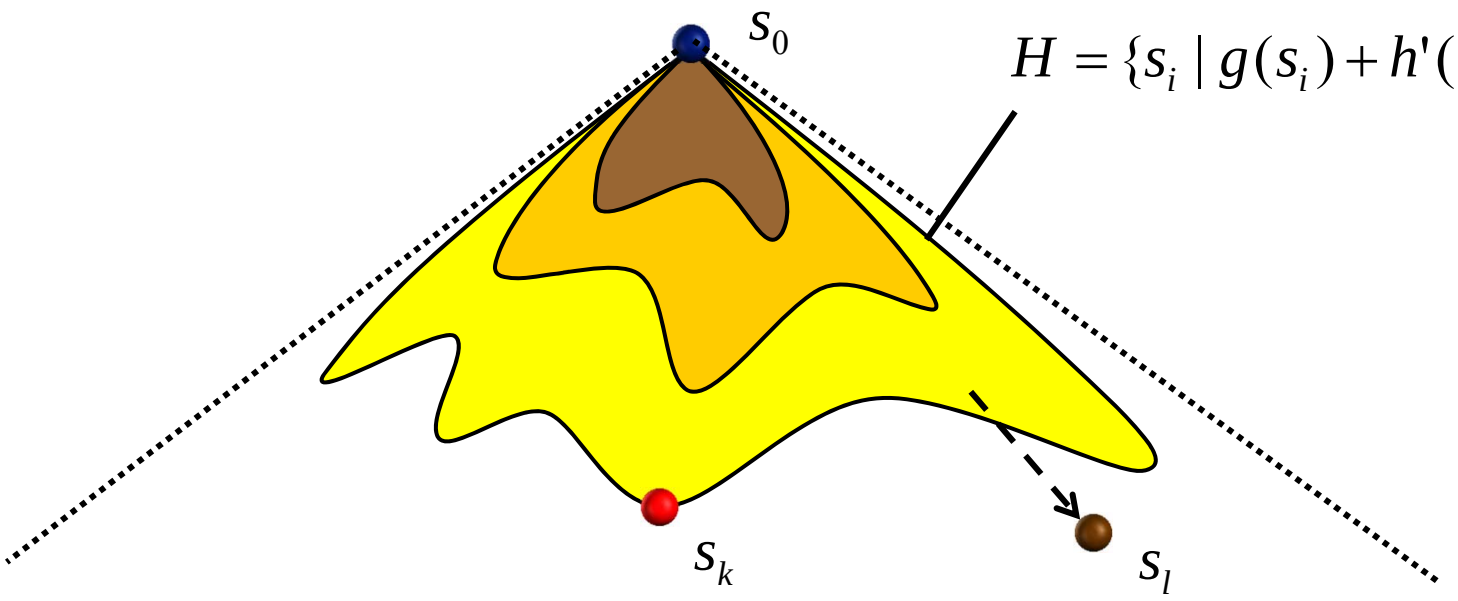
青字は関数引数



[西田 1999]

A*アルゴリズムの探索範囲と最適性

最適解を$s_{k,}$ 非最適解をs_lとする： $g(s_k) < g(s_l)$

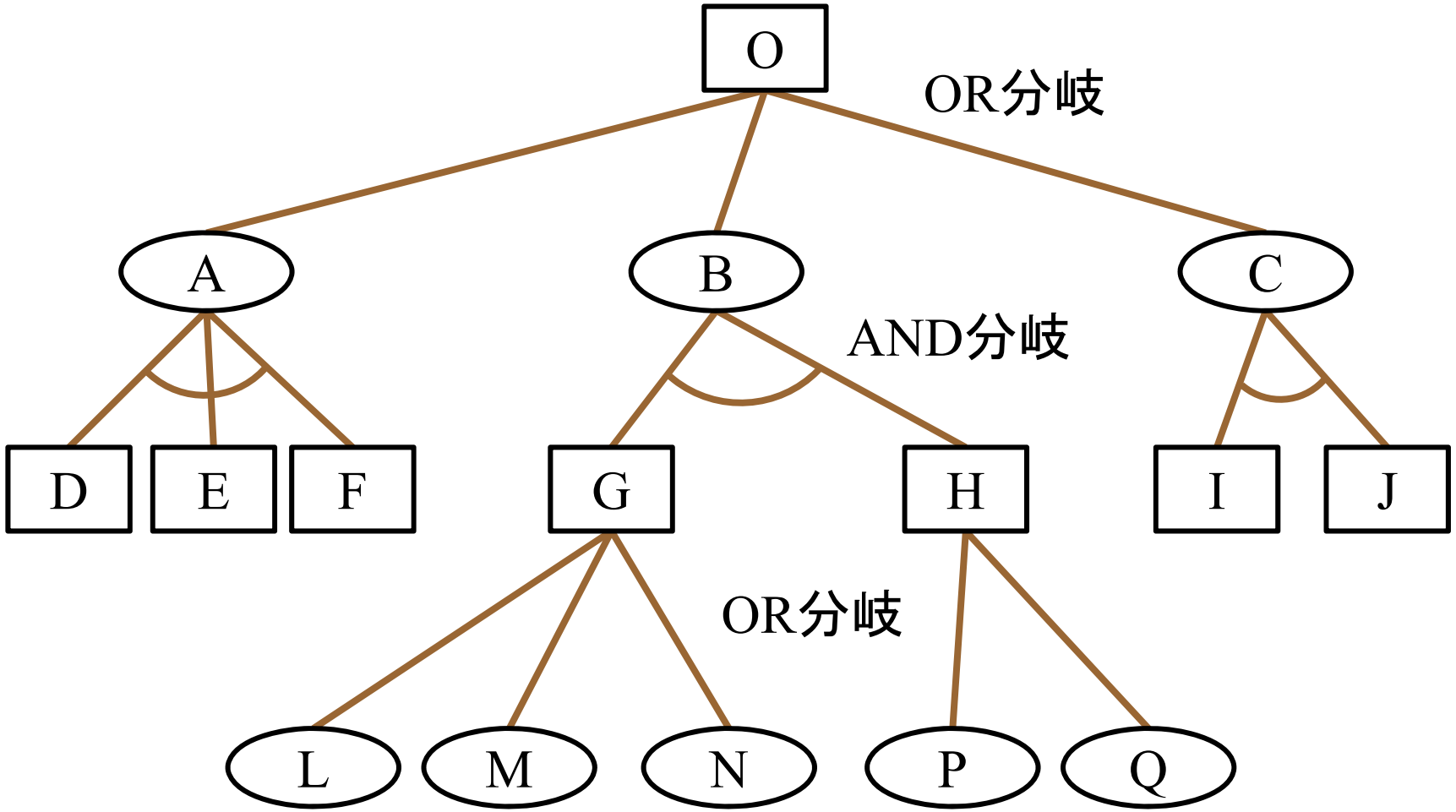


$$g(s_i) + h'(s_i) \le g(s_i) + h(s_i) = g(s_k) + h'(s_k) = g(s_k) < g(s_l) + h'(s_l) = g(s_l)$$

であるので，最適解s_kが出力される前に非最適解s_lが出力されることはない

[西田 1999]

問題分割法



[西田 1999]

問題分割法 — 不定積分

$$\int f(x)g'(x)dx = f(x)g(x) - \int f(x)g'(x)dx$$

$$\int \frac{x+1}{x^3-1}dx$$

OR分岐

$$\frac{x+1}{x^3-1} = \frac{\frac{2}{3}}{x-1} - \frac{\frac{1}{3}(2x+1)}{x^2+x+1}$$

$$\frac{\frac{x^2}{2}+x}{x^3-1} + \int \frac{\left(\frac{x^2}{2}+x\right)3x^2}{\left(x^3-1\right)^2}dx$$

$$\frac{2}{3}\int \frac{1}{x-1}dx - \frac{1}{3}\int \frac{2x+1}{x^2+x+1}dx$$

AND分岐

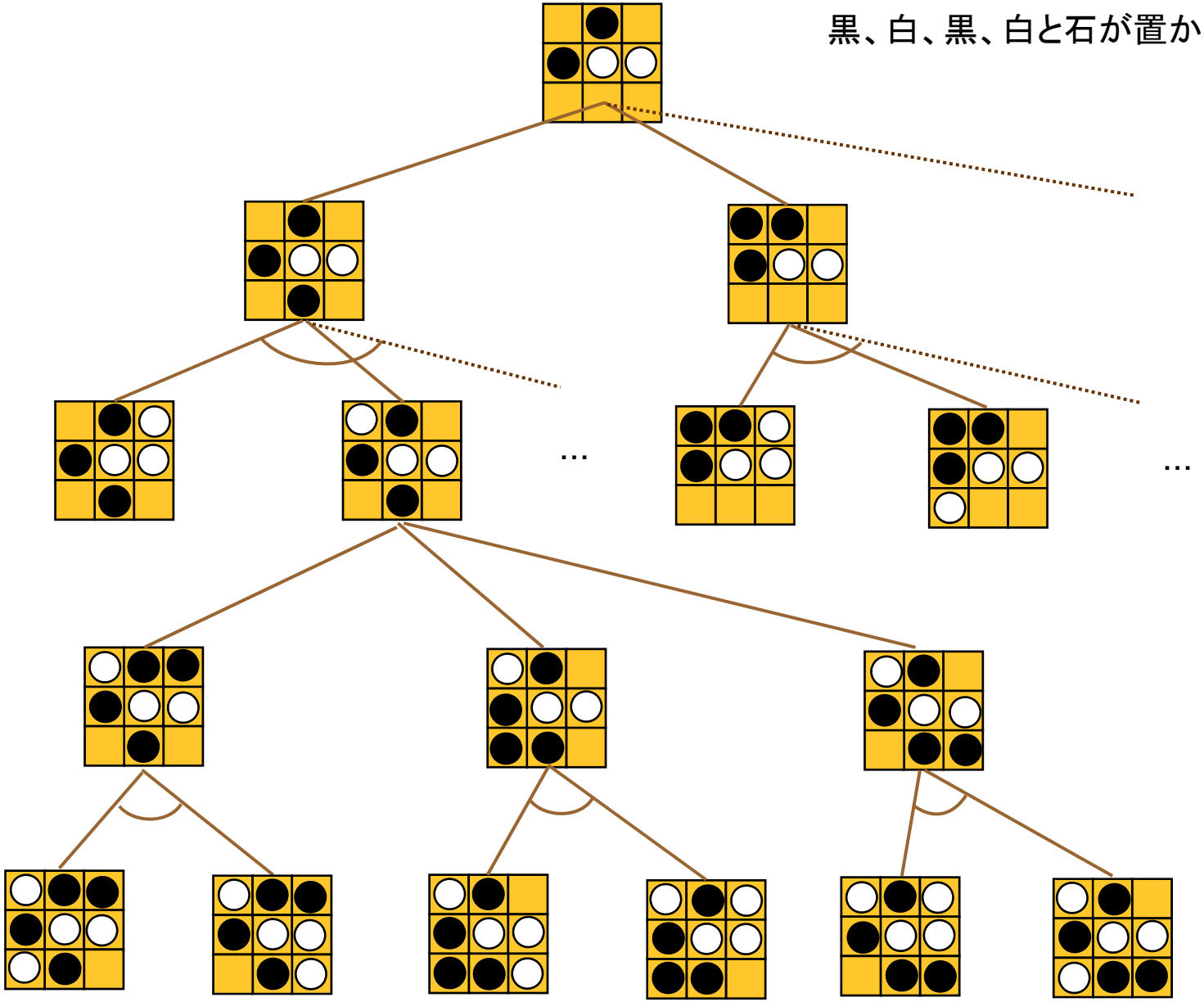
$$\int \frac{1}{x-1}dx$$

$$\int \frac{2x+1}{x^2+x+1}dx$$

[西田 1999]

問題分割法 — ゲーム

黒、白、黒、白と石が置かれ、次は黒の手番。



[西田 1999]

基本minimax法

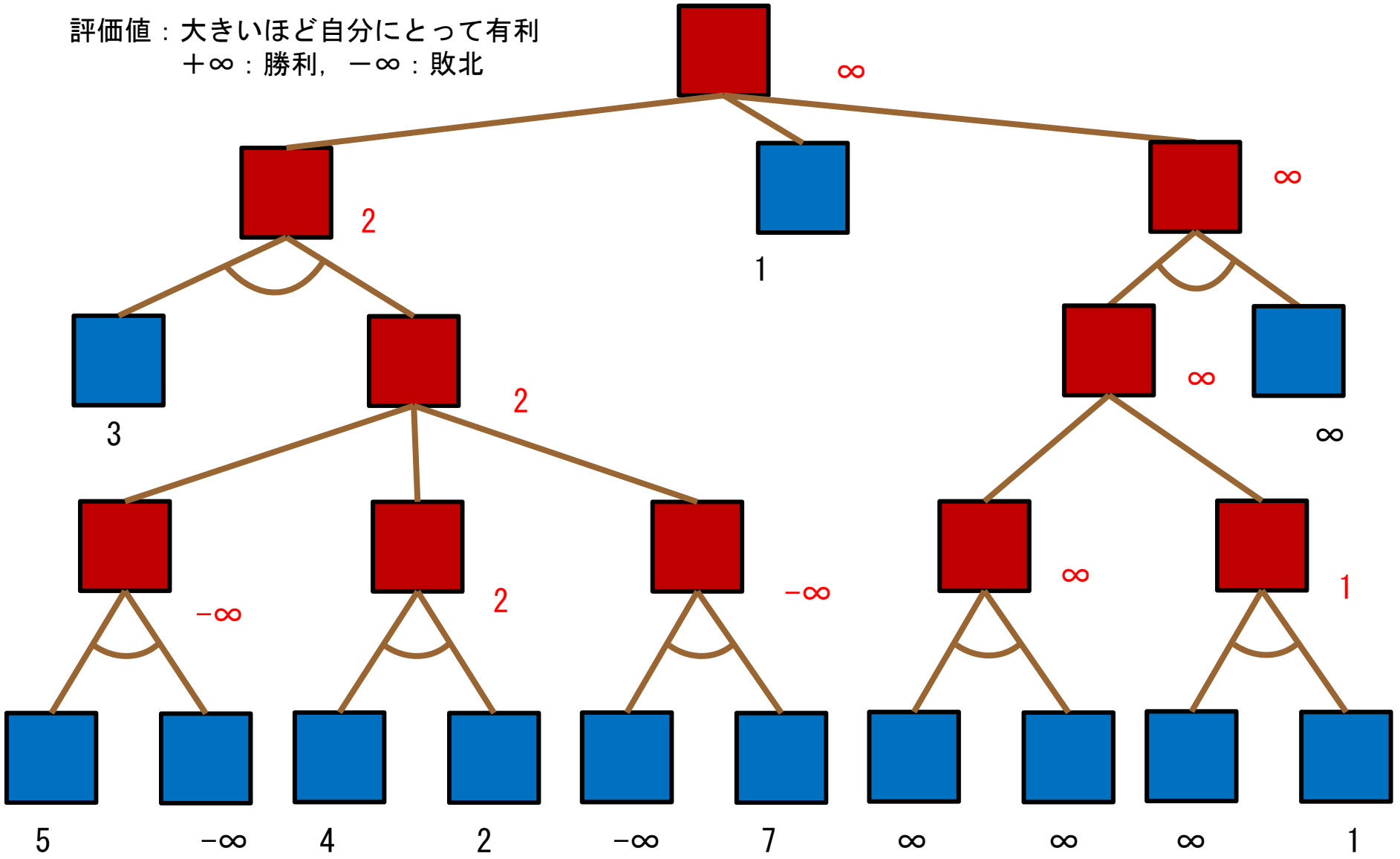
```
(defun maximizer (state max-expander max-evaluator min-expander min-evaluator)
  (let* (descendants x (result *minfty*))
     (setq descendants (apply max-expander `(,state)))
     (cond
       ((null descendants) (apply max-evaluator `(,state)))
       (t (dolist (d descendants)
            (setq x (minimizer d
                               max-expander max-evaluator
                               min-expander min-evaluator))
            (if (> x result) (setq result x)))
          result))))
```

```
(defun minimizer (state max-expander max-evaluator min-expander min-evaluator)
  (let* (descendants x (result *pinfty*))
     (setq descendants (apply min-expander `(,state)))
     (cond
        ((null descendants) (apply min-evaluator `(,state)))
        (t (dolist (d descendants)
             (setq x
                   (maximizer d
                              max-expander max-evaluator
                              min-expander min-evaluator))
             (if (< x result) (setq result x)))
           result))))
```

[西田 1999]

基本minimax法の動き

評価値：大きいほど自分にとって有利
＋∞：勝利，－∞：敗北



[西田 1999]

深さ限度付きminimaxアルゴリズム

```
(defun maximizer-depth-bound (state max-expander max-evaluator min-expander min-evaluator max-depth)
  (let* (descendants x (result *minfty*))
    (cond ((<= max-depth 0) (apply max-evaluator `(,state)))
          (t (setq descendants (apply max-expander `(,state)))
             (cond ((null descendants) (apply max-evaluator `(,state)))
                   (t (dolist (d descendants)
                        (setq x (minimizer-depth-bound d
                                                       max-expander max-evaluator
                                                       min-expander min-evaluator
                                                       (1- max-depth)))
                        (if (> x result) (setq result x)))
                      result))))))
```

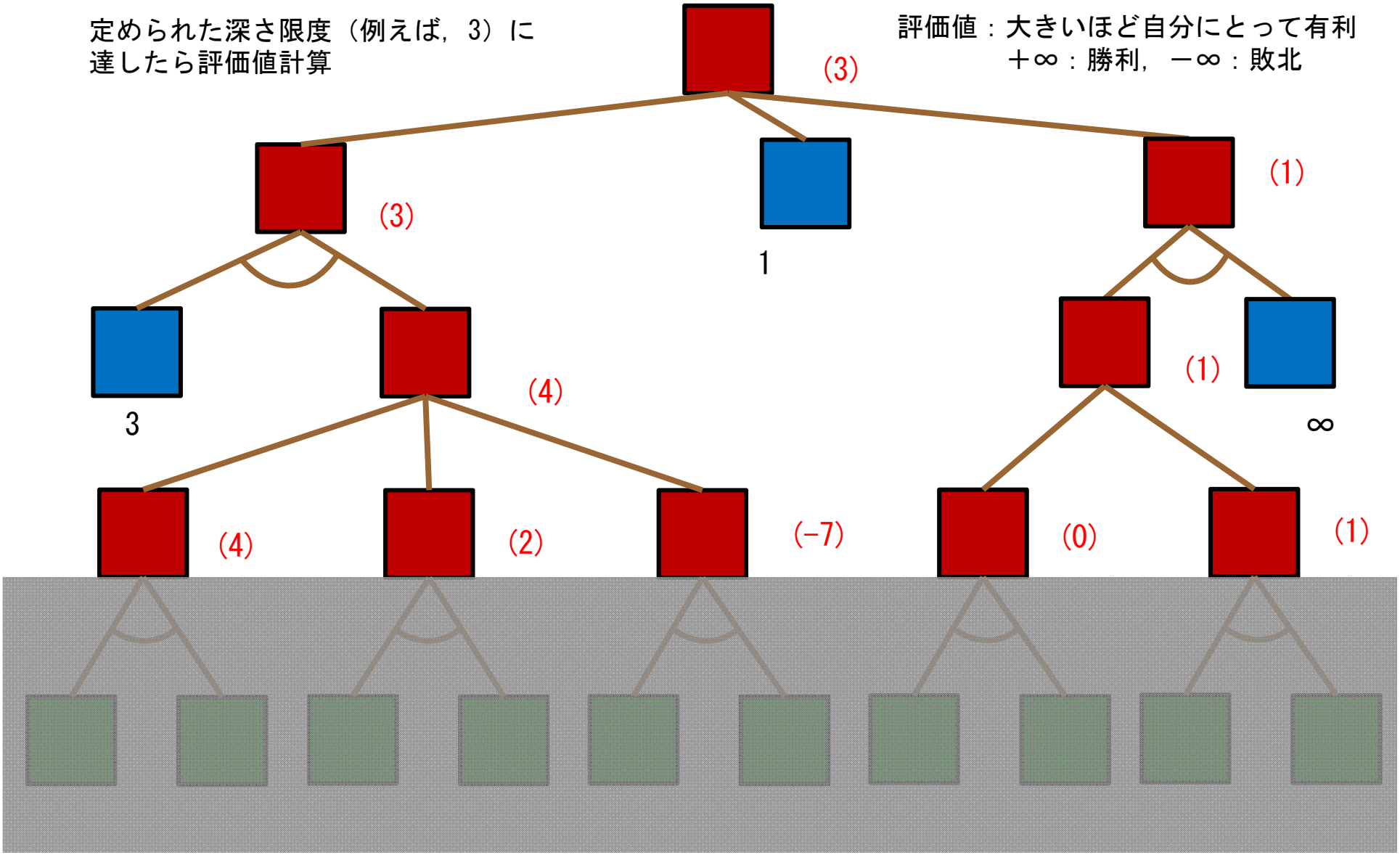
```
(defun minimizer-depth-bound (state max-expander max-evaluator min-expander min-evaluator max-depth)
  (let* (descendants x (result *pinfty*))
    (cond ((<= max-depth 0) (apply min-evaluator `(,state)))
          (t (setq descendants (apply min-expander `(,state)))
             (cond ((null descendants) (apply min-evaluator `(,state)))
                   (t (dolist (d descendants)
                        (setq x (maximizer-depth-bound d
                                                       max-expander max-evaluator
                                                       min-expander min-evaluator
                                                       (1- max-depth)))
                        (if (< x result) (setq result x)))
                      result))))))
```

[西田 1999]

深さ限度付きminimax法の動き

定められた深さ限度（例えば，3）に
達したら評価値計算

評価値：大きいほど自分にとって有利
＋∞：勝利，－∞：敗北

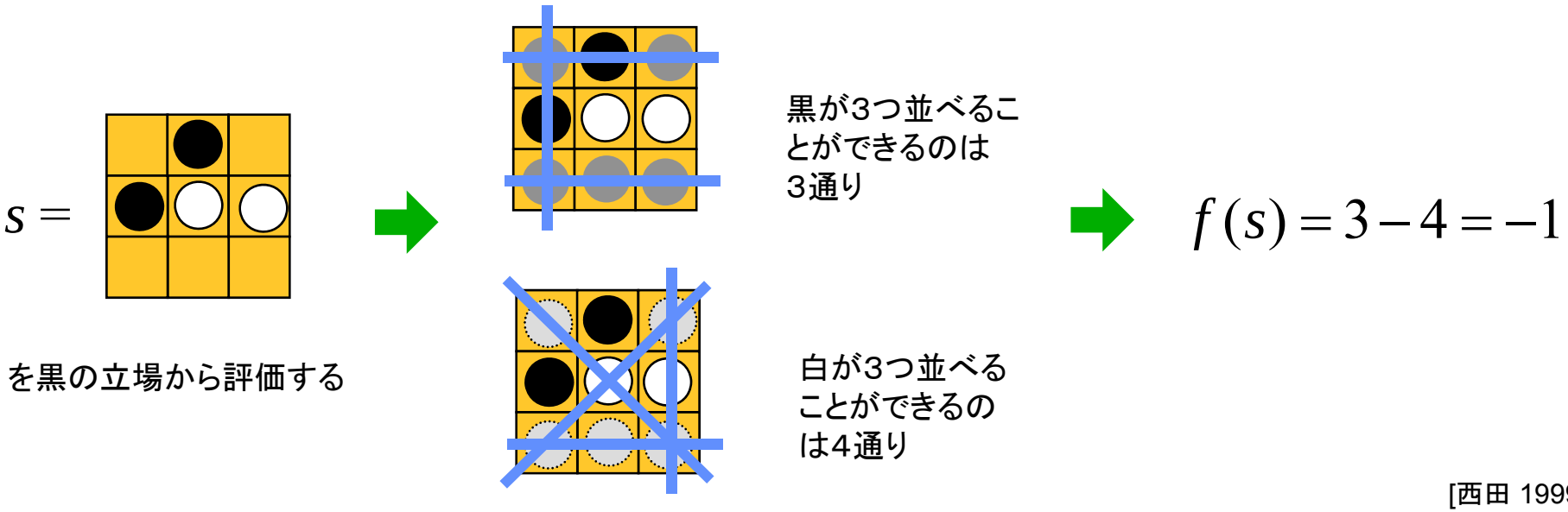


[西田 1999]

評価関数の例

3目並べに対する評価関数の一例：

$f(s)$：勝敗が自明の局面 s では，$-\infty$（相手の勝ち），0（引き分け），$+\infty$（自分の勝ち）のいずれかとする．そうでない局面 s では，自分の石を３つ連続して並べることのできる縦横斜め筋の数から，相手の石を３つ連続して並べることのできる縦横斜め筋の数を差し引いたものを値とする．

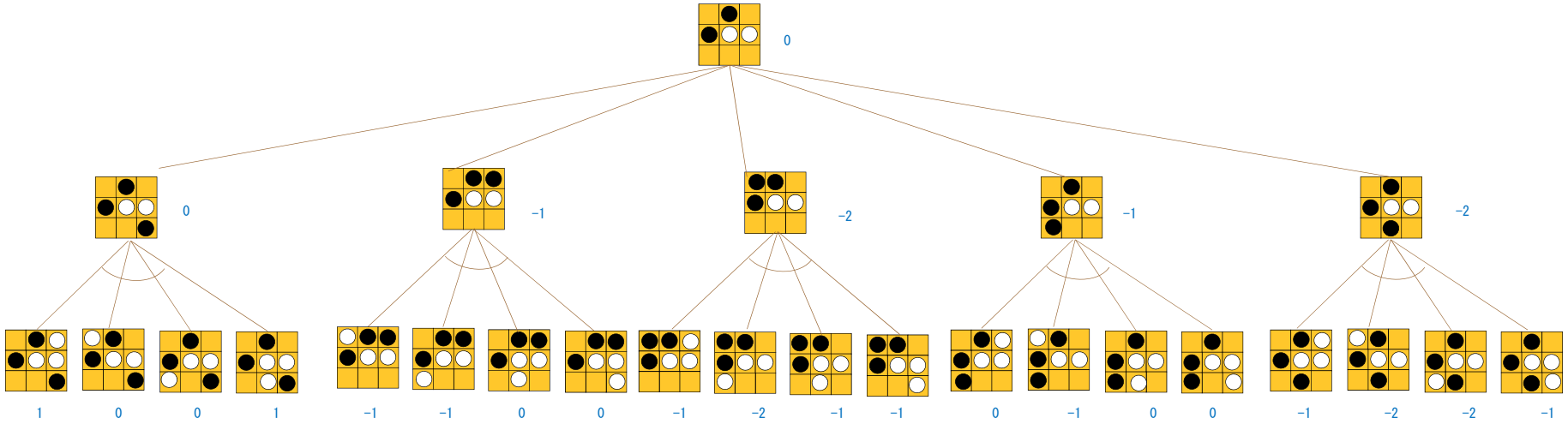


[西田 1999]

評価関数の使用例

黒、白、黒、白と石が置かれ、次は黒の手番。

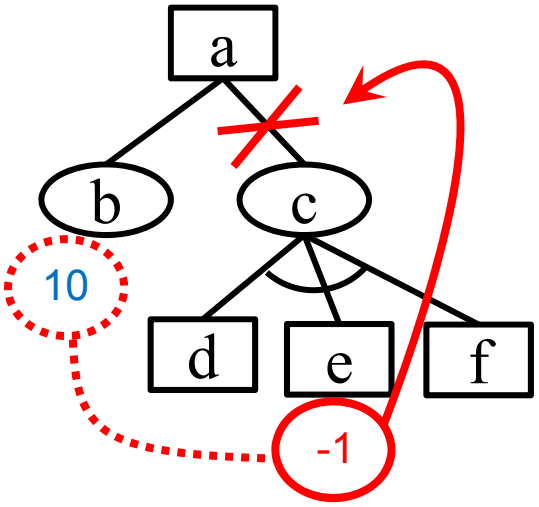
赤は真の値，（青）は評価値



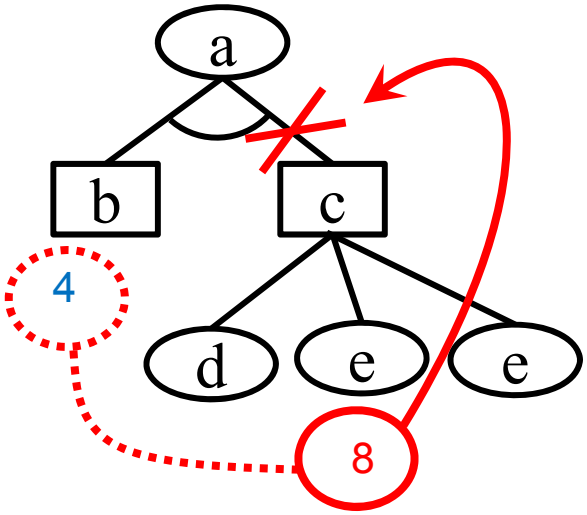
[西田 1999]

α-β探索

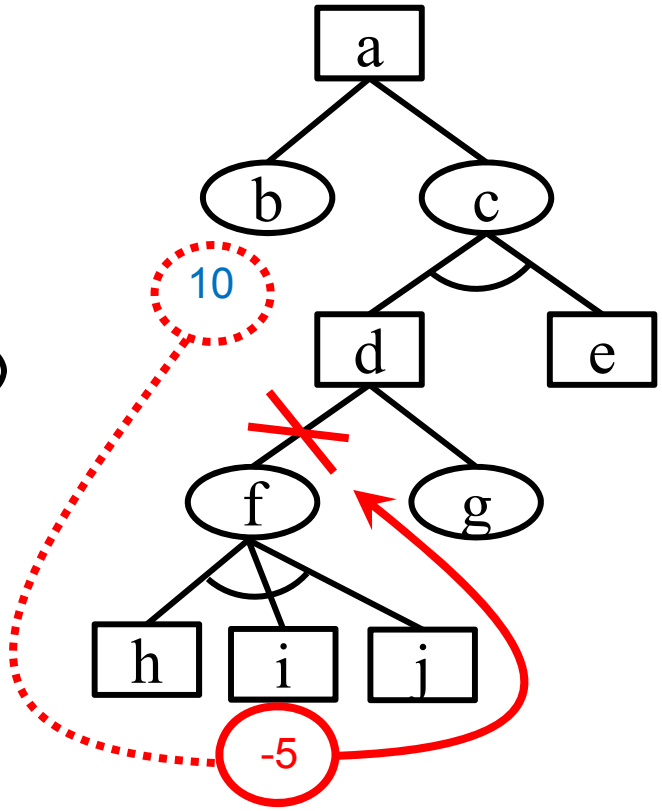
(1) α カット



(2) β カット



(3) 深いところで枝刈り

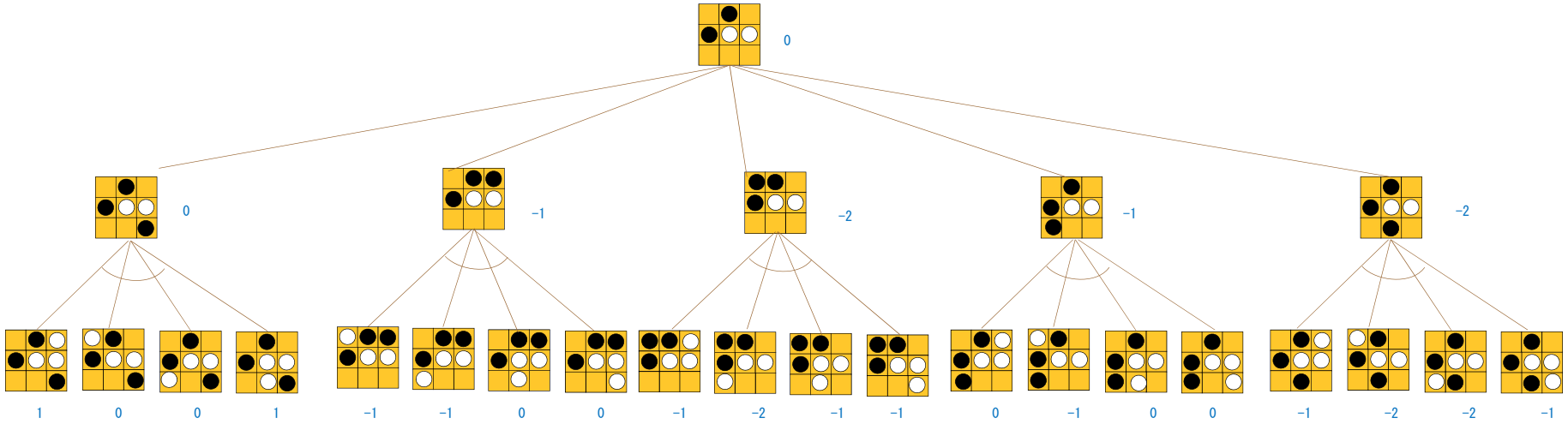


[西田 1999]

α-βカット適用前

黒、白、黒、白と石が置かれ、次は黒の手番。

赤は真の値，　（青）は評価値

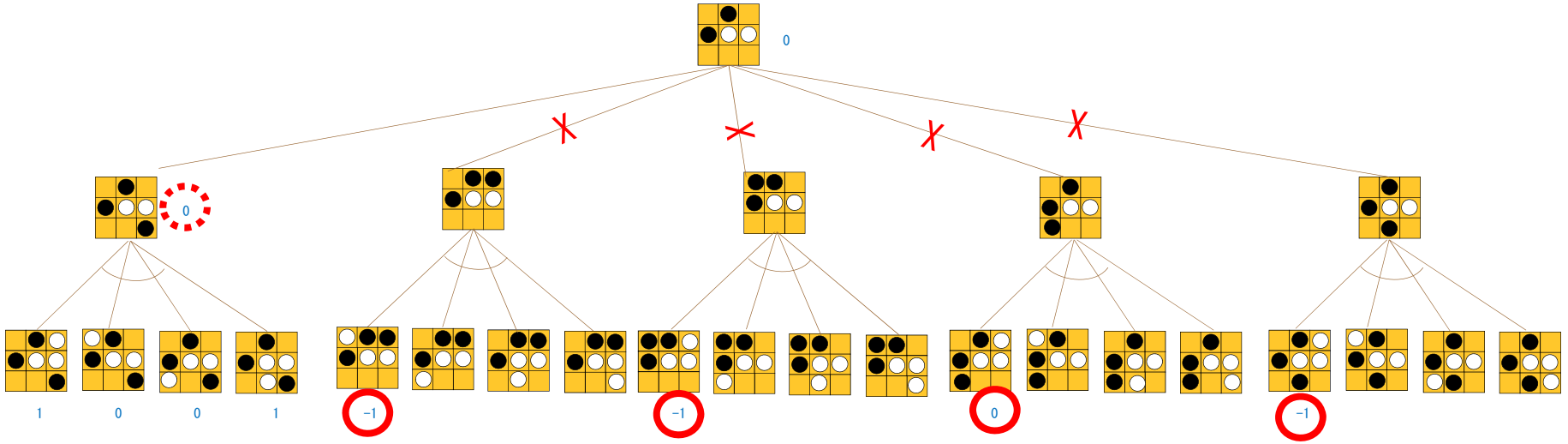


[西田 1999]

α-βカット適用後

黒、白、黒、白と石が置かれ、次は黒の手番。

赤は真の値，　（青）は評価値



[西田 1999]

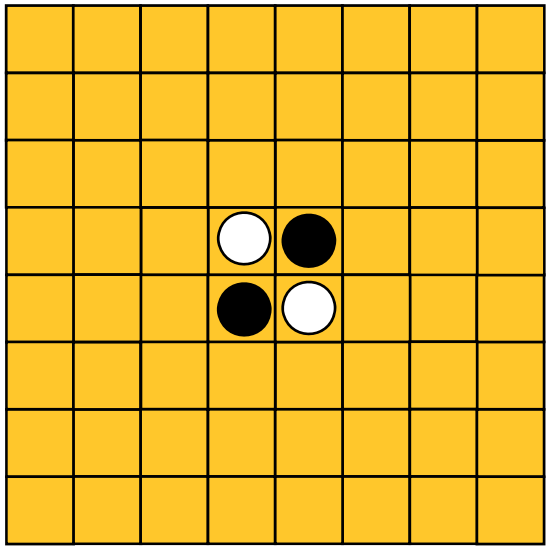
α-β探索

```
(defun alpha-depth-bound (state max-expander max-evaluator min-expander min-evaluator max-depth alpha beta)
  (let* (descendants x (result *minfty*))
    (cond ((<= max-depth 0) (apply max-evaluator `(,state)))
          (t (setq descendants (apply max-expander `(,state)))
             (cond ((null descendants) (apply max-evaluator `(,state)))
                   (t (dolist (d descendants)
                        (setq x (beta-depth-bound d
                                                  max-expander max-evaluator
                                                  min-expander min-evaluator
                                                  (1- max-depth)
                                                  result beta))
                        (cond ((null x))
                              ((<= beta x) (setq result nil) (return))
                              ((> x result) (setq result x))))
                      result))))))
```

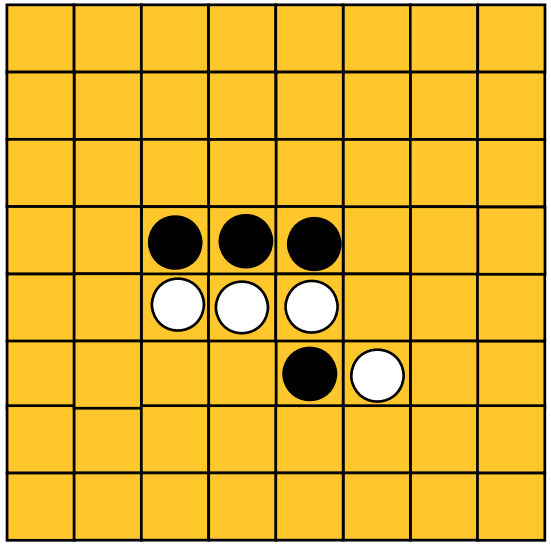
```
(defun beta-depth-bound (state max-expander max-evaluator min-expander min-evaluator max-depth alpha beta)
  (let* (descendants x (result *pinfty*))
    (cond ((<= max-depth 0) (apply min-evaluator `(,state)))
          (t (setq descendants (apply min-expander `(,state)))
             (cond ((null descendants) (apply min-evaluator `(,state)))
                   (t (dolist (d descendants)
                        (setq x (alpha-depth-bound d
                                                   max-expander max-evaluator
                                                   min-expander min-evaluator
                                                   (1- max-depth)
                                                   alpha result))
                        (cond ((null x))
                              ((>= alpha x) (setq result nil) (return))
                              ((< x result) (setq result x))))
                      result))))))
```

[西田 1999]

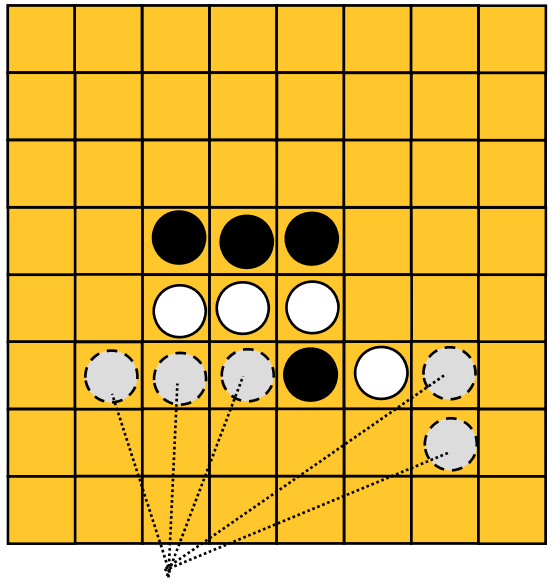
応用　オセロゲーム



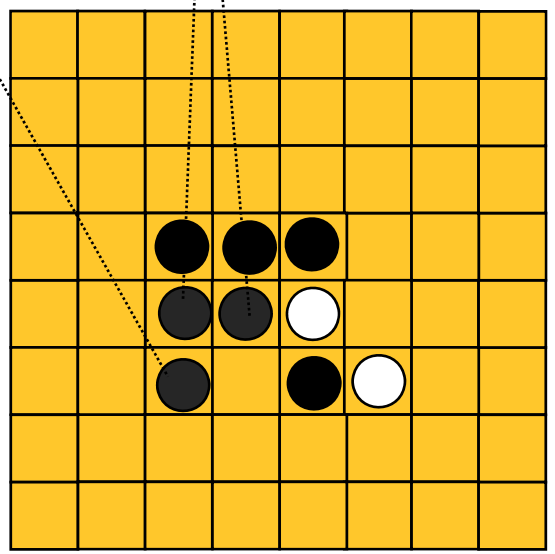
(1)　最初の盤面



(2a)　ある盤面(次は黒)



(2b)　黒石の置けるところ



(2c)　次の盤面

置かれた黒石によって黒に反転した白石

黒石が置かれたところ

応用　オセロゲーム

120	-20	20	5	5	20	-20	120
-20	-40	-5	-5	-5	-5	-40	-20
20	-5	15	3	3	15	-5	20
5	-5	3	3	3	3	-5	5
5	-5	3	3	3	3	-5	5
20	-5	15	3	3	15	-5	20
-20	-40	-5	-5	-5	-5	-40	-20
120	-20	20	5	5	20	-20	120

評価関数の例　[Novig 1992]