インテリジェントシステム

#4 敵対的探索とゲーム adversarial search

信州大学工学部電子情報システム工学科
丸山稔

ゲーム木の探索：２プレイヤー・ゼロサムゲーム（zero-sum games）

2人のプレーヤーによるゲームを考える：
　2人⇒MAX/MINと呼ぶ　（片方はMAXを目指し、片方はMINを目指す）
　ゲーム終了時に勝者←points, 敗者←penalties
　ゼロサム・ゲーム（zero-sum games）← 片方に良いことはもう片方にとって悪い（足したら０）

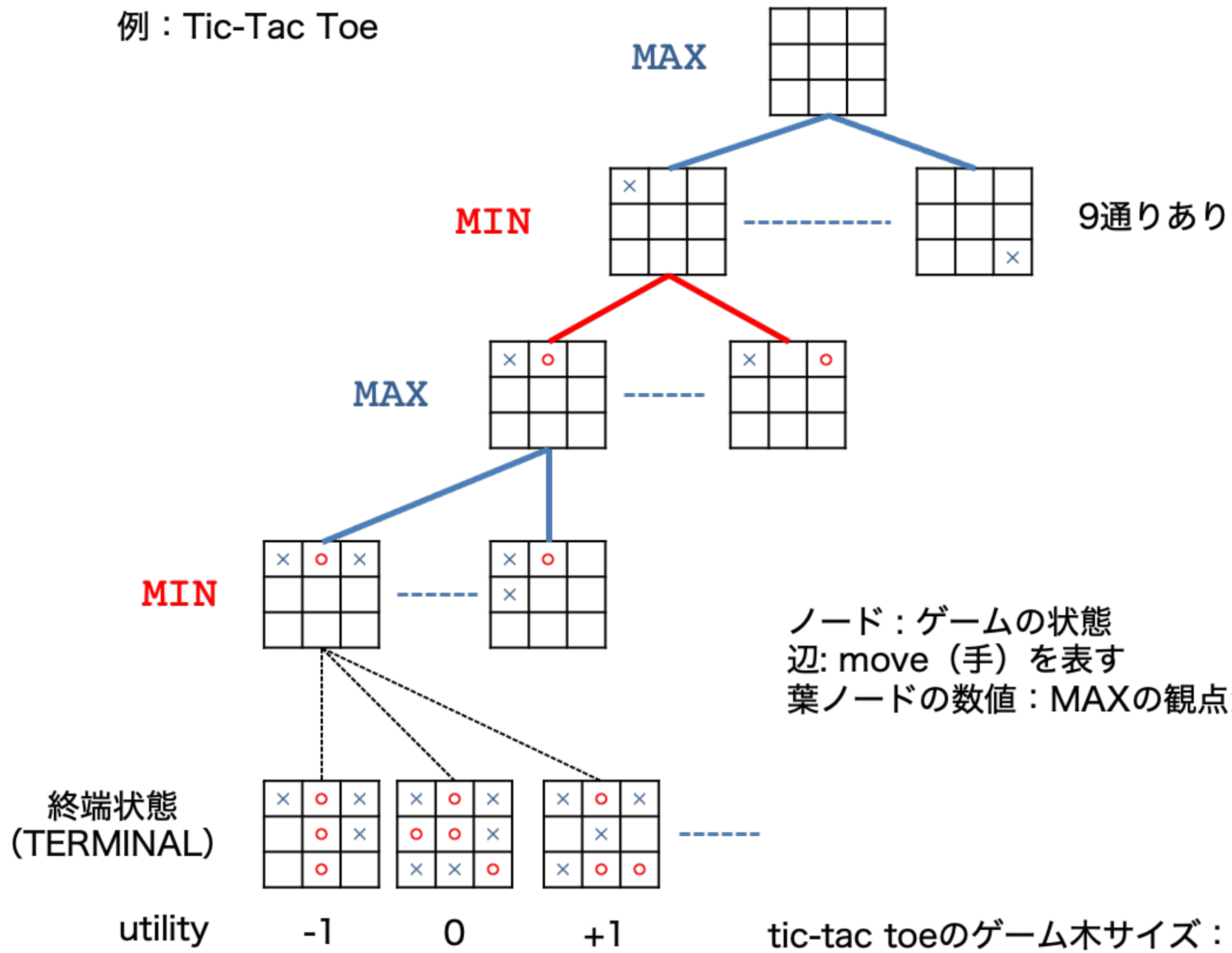
ゲームの形式的定義：以下の要素で定義される

- s_0：初期状態
- `TO-MOVE(s)`：状態sにおいて手（move）を打つ順番のプレイヤー
- `ACTIONS(s)`：状態sにおいて可能な全ての手（move）の集合
- `RESULT(s,a)`：遷移モデル状態sにおいて行動aを選択した結果もたらされる状態
- `IS-TERMINAL(s)`：終端状態テスト…ゲームが終了なら`true`、そうでなければ`false`
- `UTILITY(s,p)`：効用（utility）関数（または目的関数、payoff関数などとも呼ばれる）
  →プレーヤーpが状態sで終了したゲームの最終的数値（報酬）を表す

初期状態, 行動集合（Actions）,後続状態（Succ） ⇒ゲーム木（game tree）を定義する

ゲーム木：ノード→ゲームの状態, 辺→手（move）を表す

例：Tic-Tac Toe



ノード：ゲームの状態
辺: move（手）を表す
葉ノードの数値：MAXの観点からのutility値

tic-tac toeのゲーム木サイズ：比較的小$\leq 9! = 362880$
チェス⇒10^{40}ノード以上

ゲームの場合、通常の探索問題とは異なる

・通常の探索問題：最適解⇒初期状態からゴールまでの行動系列

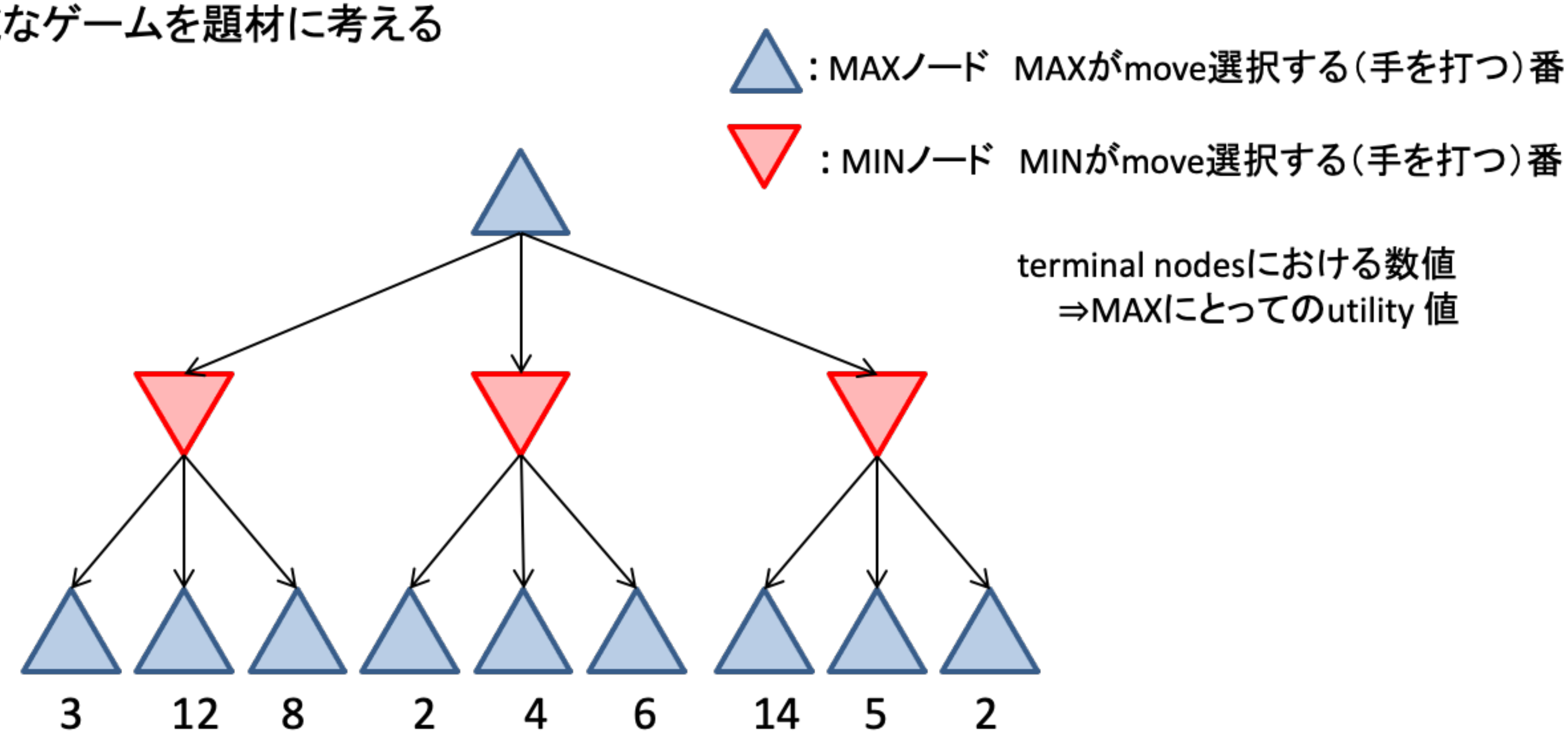
⇔　ゲームの場合：相手プレーヤーがどのような行動を取るか分からない

∴解は固定された行動系列ではなく、戦略（strategy）またはポリシー（policy）

写像：状態 →その状態における最適move

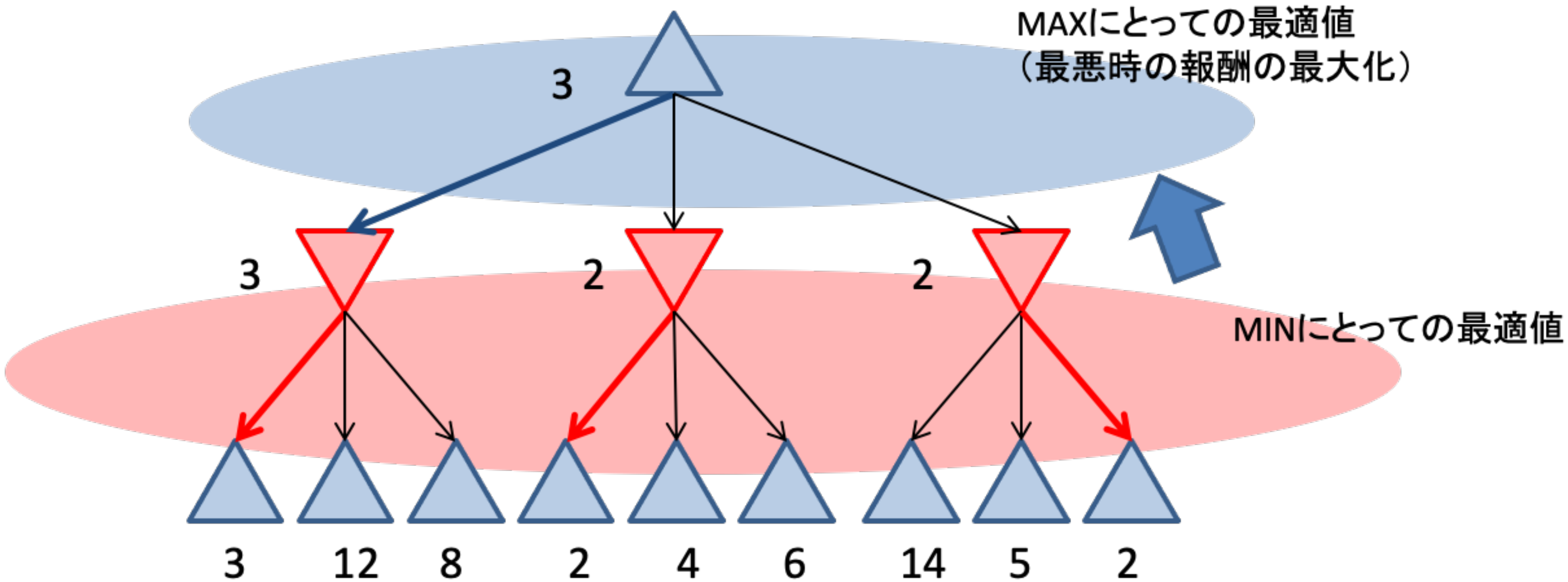
MAXの取るべき手？⇒可能な全てのMINの手によってもたらされる可能性のある状態を考慮する

単純なゲームを題材に考える



ミニマックス戦略（Min-Max Strategies）

・相手が最適手（最善手）を取ることを想定＆自分も最適手を取ることを想定
・最悪ケースの報酬の中で最善の値が得られるような手を取る
⇒　終端ノード（ゲーム終了状態）：効用関数の値（utility value）
　　終端状態以外のノード：minimax 値



MINIMAX(n)：MAX,MINともに最適手を取ったときの（MAXにとっての）utility値

$$
\mathtt{MINIMAX(s)} = \begin{cases} \mathtt{UTILITY(s, MAX)} & \textbf{if } \mathtt{IS\text{-}TERMINAL(s)} \\ \max_{a \in Actions(s)} \mathtt{MINIMAX(RESULT(s, a))} & \textbf{if } \mathtt{TO\text{-}MOVE(s) = MAX} \\ \min_{a \in Actions(s)} \mathtt{MINIMAX(RESULT(s, a))} & \textbf{if } \mathtt{TO\text{-}MOVE(s) = MIN} \end{cases}
$$

```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move


function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v, move ← -∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state,a))
        if v2 > v then
            v, move ← v2, a
    return v, move


function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v, move ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state,a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

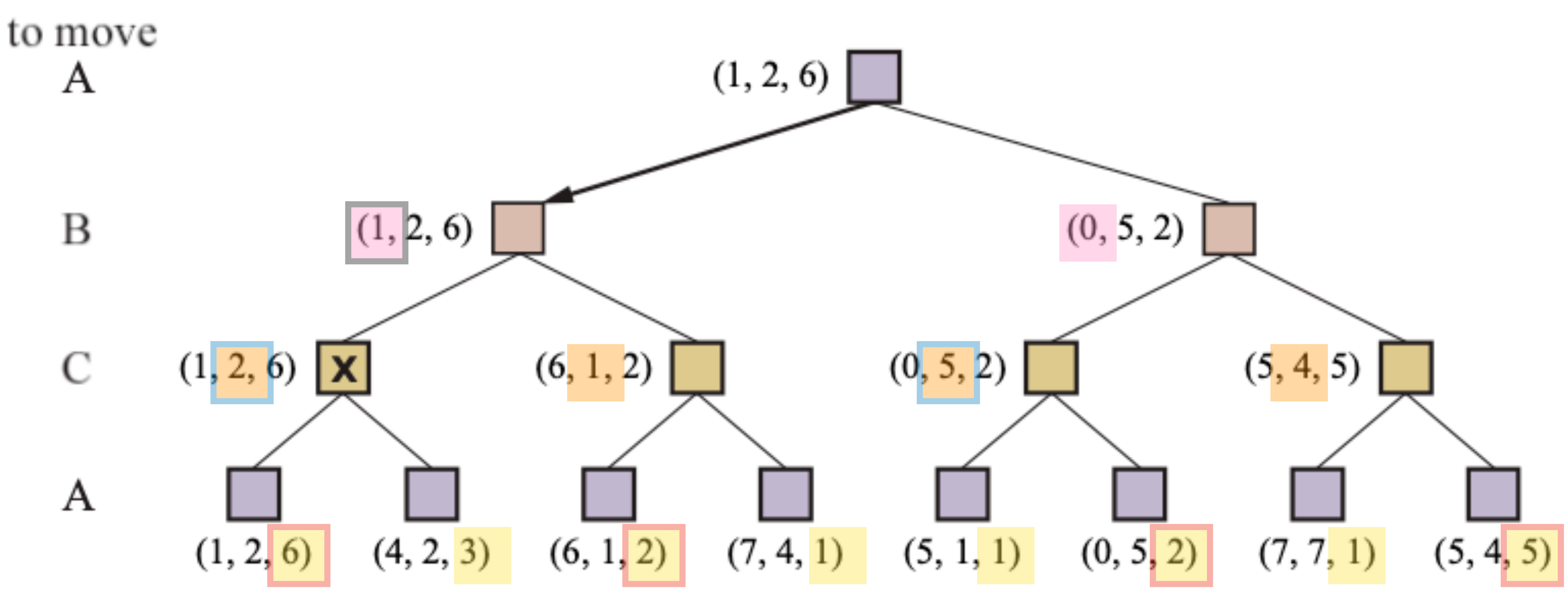
ミニマックスアルゴリズム⇒ゲーム木の深さ優先探索

深さ最大値m, 各ノードでb個の手が可能とすると

⇒ 計算時間 $O(b^m)$

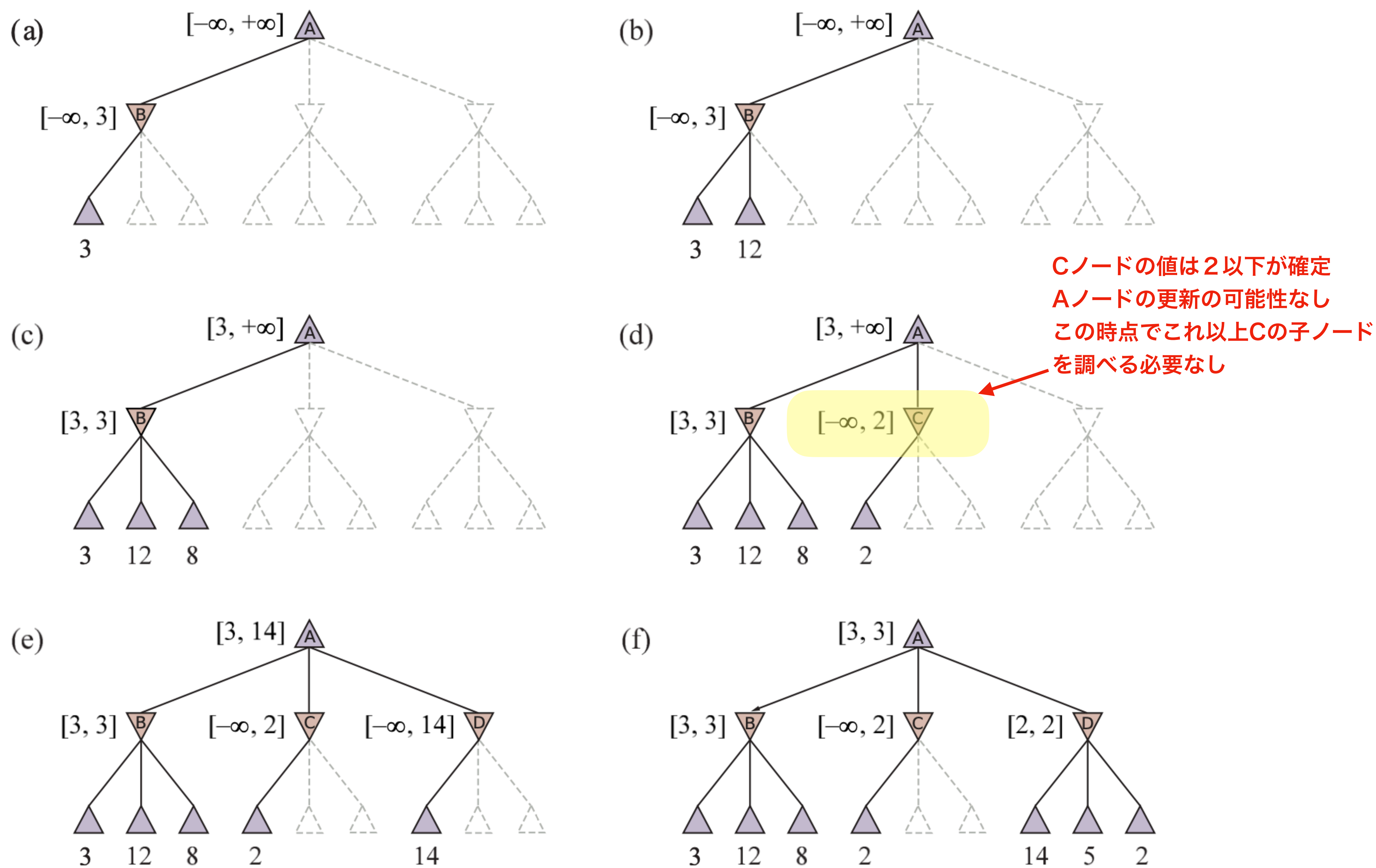
現実の(状態空間が大きな)ゲームに用いることは現実的でない
但し、現実的手法の基盤や解析の基礎として有用

２人以上のプレーヤーを許容するゲームの場合
各ノード：これまで１つの値が対応⇒ベクトルに変更
Ex. players A,B,C ⇒ベクトル値
終端状態では各playerの立場でのutility値を示す
各playerは自分のutility値が最大になるように手を選択する



S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

アルファ・ベータ枝刈り（alpha-beta pruning）



S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

```
function ALPHA-BETA-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state, -∞, +∞)
   return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) ペア
   if game.IS-TERMINAL(state) then return game.UTILITY(state,palyer), null
   v← -∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state,a),α, β)
      if v2 > v then
         v, move ← v2,a
         α ← MAX(α, v)
      if v >= β then return v, move
   return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) ペア
   if game.IS-TERMINAL(state) then return game.UTILITY(state,palyer), null
   v← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state,a),α, β)
      if v2 < v then
         v, move ← v2,a
         β ← MIN(β, v)
      if v <= α then return v, move
   return v, move
```

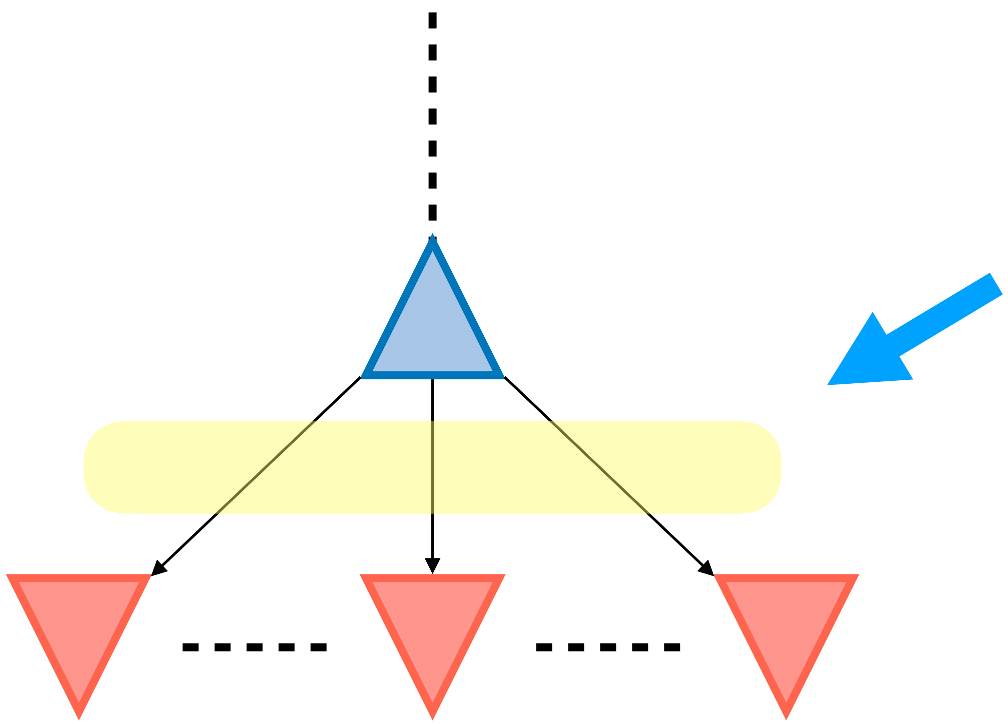
アルファ・ベータ枝刈り（alpha-beta pruning）

ノードnを含む経路において見つかった

α：MAXにとって、これまでの最良値（最大値）⇒最終的にはこれより大

β：MINにとって、これまでの最良値（最小値）⇒最終的にはこれより小

MAXノードの場合：



子ノード（MINノード）の値を評価

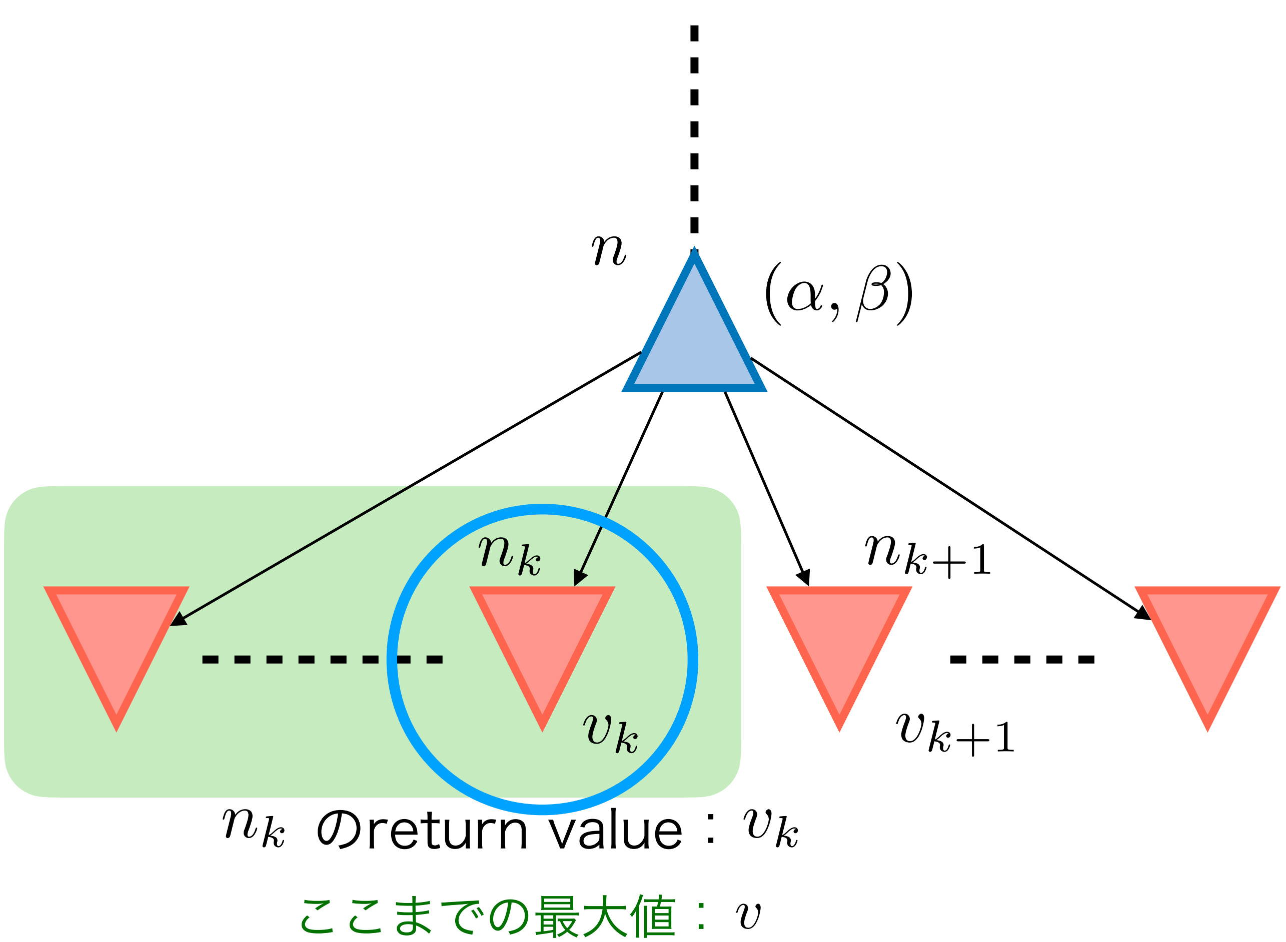
return valueがβより大きければ、このMAXノードの出力はβより大きくなる

⇒　最終的にはこの値は生き残らない

子ノードの評価時点での無駄な探索を避けるために（α,β）値（MAXノードの場合はα値を必要に応じて更新

MAXノードにおける処理

```
function MAX-VALUE(game, state, α, β) returns a (utility, move) ペア
   if game.IS-TERMINAL(state) then return game.UTILITY(state,palyer), null
   v← -∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state,a),α, β)
      if v2 > v then
         v, move ← v2,a
         α ← MAX(α, v)
      if v >= β then return v, move
   return v, move
```



子ノードであるMINノードの値を調べていく

→MAXノードなので最大値を返すのが基本

n_k の評価値⇒ v_k

ここまでの最大値：v

v ：βより大⇒ノードnの出力はβを超える（α,β）の範囲を超えるので、この結果が最終的に生き残ることはない⇒**探索打ち切り**

これから探索する n_{k+1}

$v_{k+1} < v$　となればMAXノードnの出力とはならない

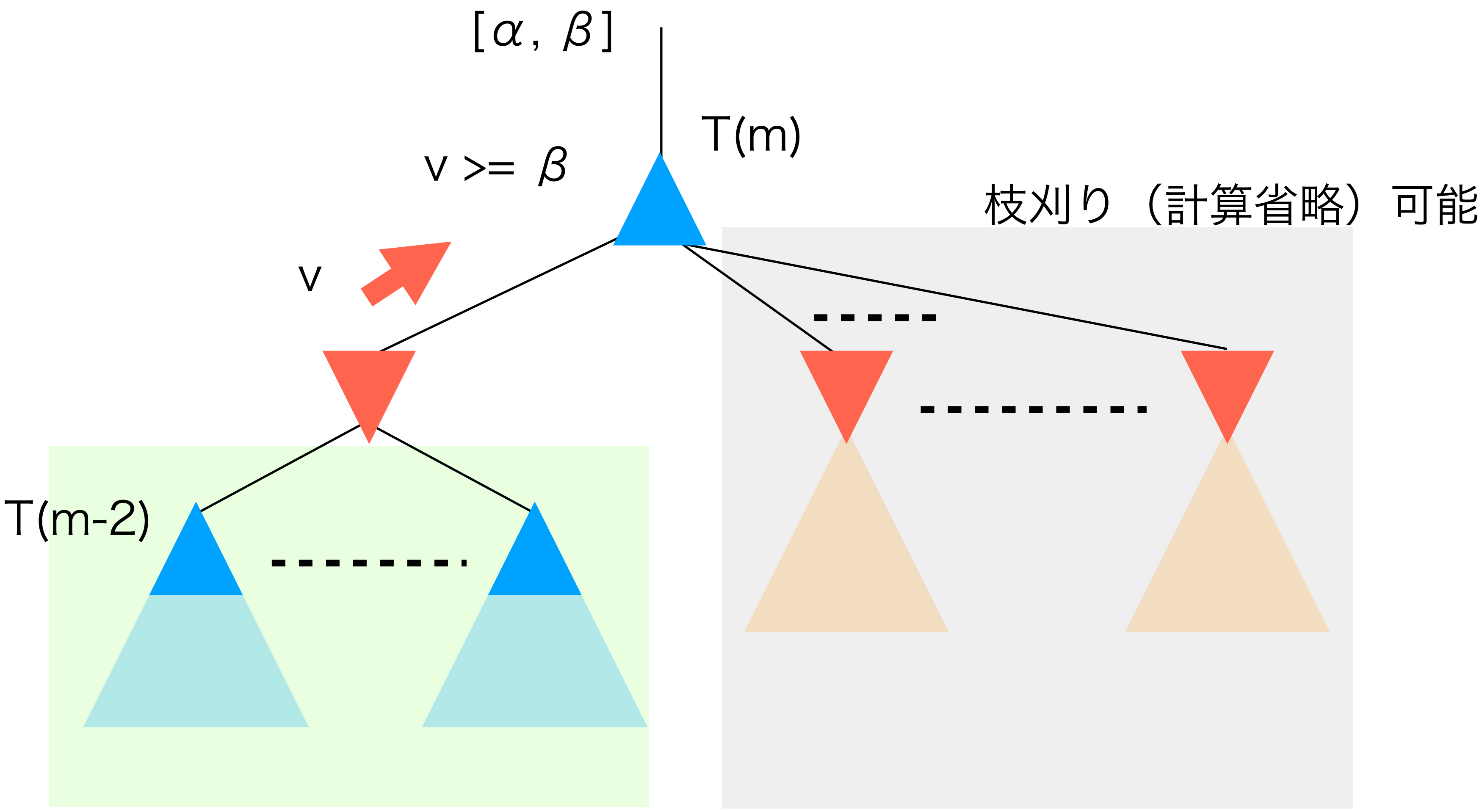
$v_{k+1} > v$　のときvを更新…さらに

$$\alpha = \max\{\alpha, v\}$$

とα値更新して子ノードの探索続行

アルファ・ベータ枝刈り（alpha-beta pruning）

最良の場合どの程度枝刈りできる？



$$T(m) = bT(m-2) + 1$$

$$T(m) = bT(m-2) + 1 \quad \Rightarrow \quad T(m) = b^k T(m-2k) + b^{k-1} + b^{k-2} + \cdots + 1$$

$$\Rightarrow \quad T(m) = O(b^{\frac{m}{2}})$$

ヒューリスティックα-β木探索（heuristic alpha-beta tree-search）

計算時間に制限があるとき⇒ 探索を早期に切り上げヒューリスティック評価関数を適用

⇒ UTILITYをEVALで置き換え

終端テスト（terminal test）をcutoffテストで置き換え

↓

終端ならばtrueとする…それ以外の動作には自由度

↓

深さと状態評価により探索を打ち切るか決定

$$\text{H-MINIMAX (s,d)} = \begin{cases} \text{EVAL}(\text{s}, \text{MAX}) & \textbf{if } \text{IS-CUTOFF(s,d)} \\ \max_{a \in Actions(s)} \text{H-MINIMAX(RESULT(s,a),d+1)} & \textbf{if } \text{TO-MOVE(s) = MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX(RESULT(s,a), d+1)} & \textbf{if } \text{TO-MOVE(s) = MIN} \end{cases}$$

ヒューリスティック評価関数：**EVAL(s,p)**

→ 状態sにおけるプレイヤーpへの**UTILITY**期待値の推定値を返す

終端状態では**EVAL(s,p) = UTILITY(s,p)**

非終端状態では勝ち/負けの間の値 **UTILITY(loss, p) <= EVAL(s,p) <= UTILITY(win, p)**

…となるのは当然として…

他に求められる性質としては：

1. 計算時間がそれほど大きくない
2. 勝つチャンスと高い相関を持つ

状態の"特徴"から評価値を求めることがよく行われる⇒ $\text{EVAL(s)} = \sum_{i=1}^{n} w_i f_i(s)$ 　 $f_i(s)$：特徴量

探索の打ち切り

α-β 探索（ALPHA-BETA-SEARCH）におけるIS-TERMINALの部分を以下で置き換え

（探索打ち切りの際UTILITY評価ではなくヒューリスティック関数 EVALを呼び出す）

```
if game.IS-CUTOFF(state, depth) then return game.EVAL(state, player), null
```

さらに、再起呼び出しの際にdepthが更新されるようにする

→　設定されたdepthに関するしきい値 dを超過したらIS-CUTOFFはtrue

またはiterative deepening（反復深化）を用いる

→ 時間切れのときには深さ最大の探索の結果を返す

Forward pruning

α-β 枝刈り⇒ 最終結果に影響を与えない枝を切り捨て

…それに対し…↔ forward pruning : poor moveと見える・判断できる枝を切る

↓

最終的には良好な結果が得られる可能性

計算時間は節約できる

forward pruningの例：ビームサーチ（ beam search）

評価関数が良い値を持つn個のベストmoveのみを考慮の対象とする

モンテカルロ木探索（MCTS, Monte-Carlo Tree Search）

ヒューリスティックα-β探索の弱点：
・枝分かれの数が多い場合⇒数個先の手までに限定される
・性能の良い評価関数を得るのは難しい

⇒ MCTS（Monte-Carlo Tree Search）が用いられる
状態の価値（value）の推定…ヒューリスティック評価関数は使わない
その状態から開始した多数回のシミュレーション結果の平均値により評価
↓
playout, rollout と呼ばれる

シミュレーション中で手をどのように選択する？
ランダム？→プレイヤーがランダムにプレイした場合に相当…多くの場合うまくいかない

プレイアウト方策（palyout policy）が必要→囲碁等ではニューラルネットを用いてself-playから学習

プレイアウト方策が与えられたとき以下を決定する必要あり
→ どのポジションからプレイアウトを開始するか
各ポジションにどの程度の回数のpalyoutsを割り当てるか

モンテカルロ木探索（MCTS, Monte-Carlo Tree Search）

シミュレーションの実行　←　選択方策（selection policy）が必要

↓

計算資源をゲーム木の重要な箇所に選択的に割り当て

探索＆活用（exploration & exploitation）のバランスが重要

・exploration … まだあまり調べていない状態を調べる

・exploitation … 過去のplayoutsでうまく行くことが示された状態

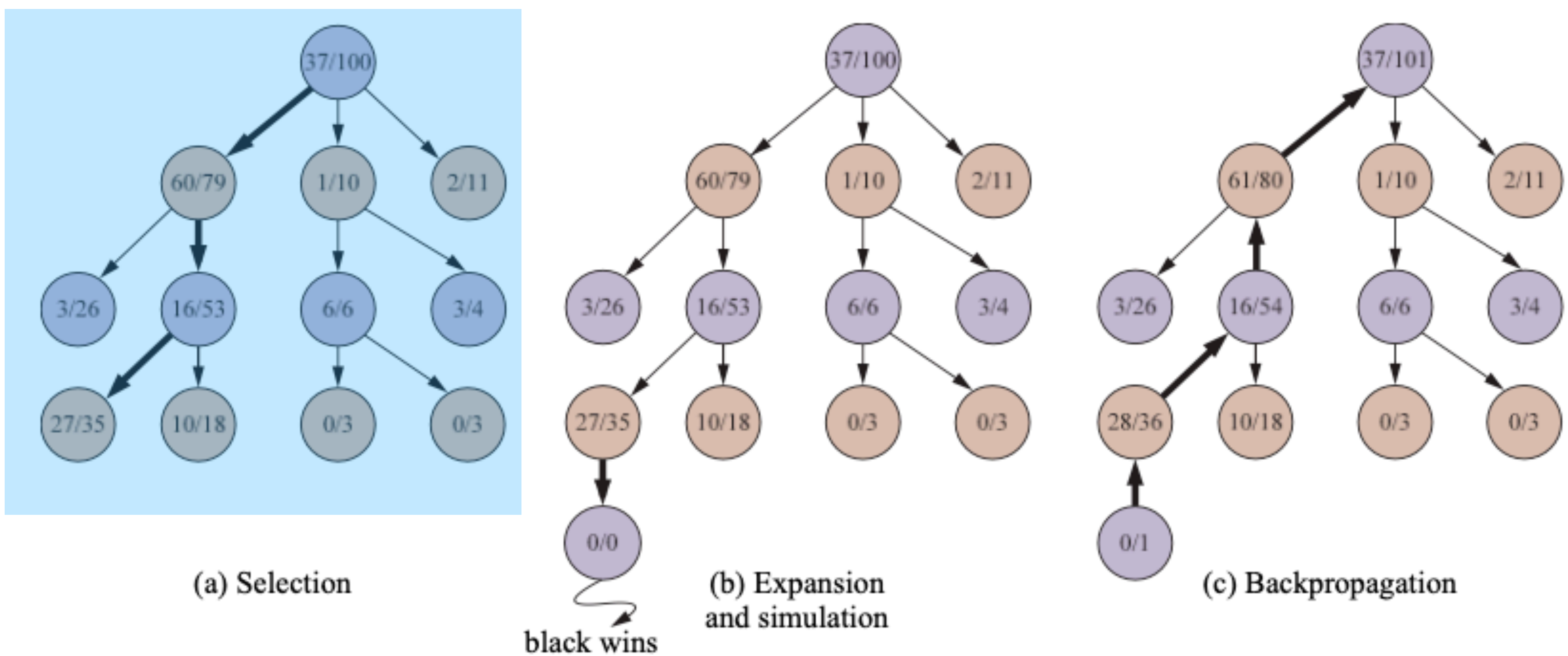
→　価値をさらに精密に評価

MCTS：探索木を保持→次の４ステップを繰り返して木を成長させる

- 選択（selection）
  rootノードから開始して、選択方策に従って手を選択し、後続ノードに進む
  これを繰り返して葉ノードに到達
- 展開（expansion）
  選択したノードに新規子ノードを生成して探索木を成長させる
- シミュレーション（simulation）
  新規子ノードからpalyout policyに従って手を選択していく
  →これらの手（経過）は探索木に記録されない
- 逆伝搬（back-propagation）
  シミュレーション結果を用いてルートまでの経路を逆に辿る
  → 各ノードの勝率等を更新

モンテカルロ木探索（MCTS, Monte-Carlo Tree Search）

palayers（名称）はwhiteとblack…最初white, 次はblack …と順次手を選択していく
selection:探索木のルートから開始して葉ノードへ
root：ここから開始したplayoutについてwhiteが100回中37回勝利
太線が選択された手を表す→blackが79回の palyoutsのうち60回勝利した手を選択
…3つの選択肢中勝率最大…exploitationの例
…2/11を選択しても悪くない（かも）…exploration
選択を続け27/35の葉ノードに至る



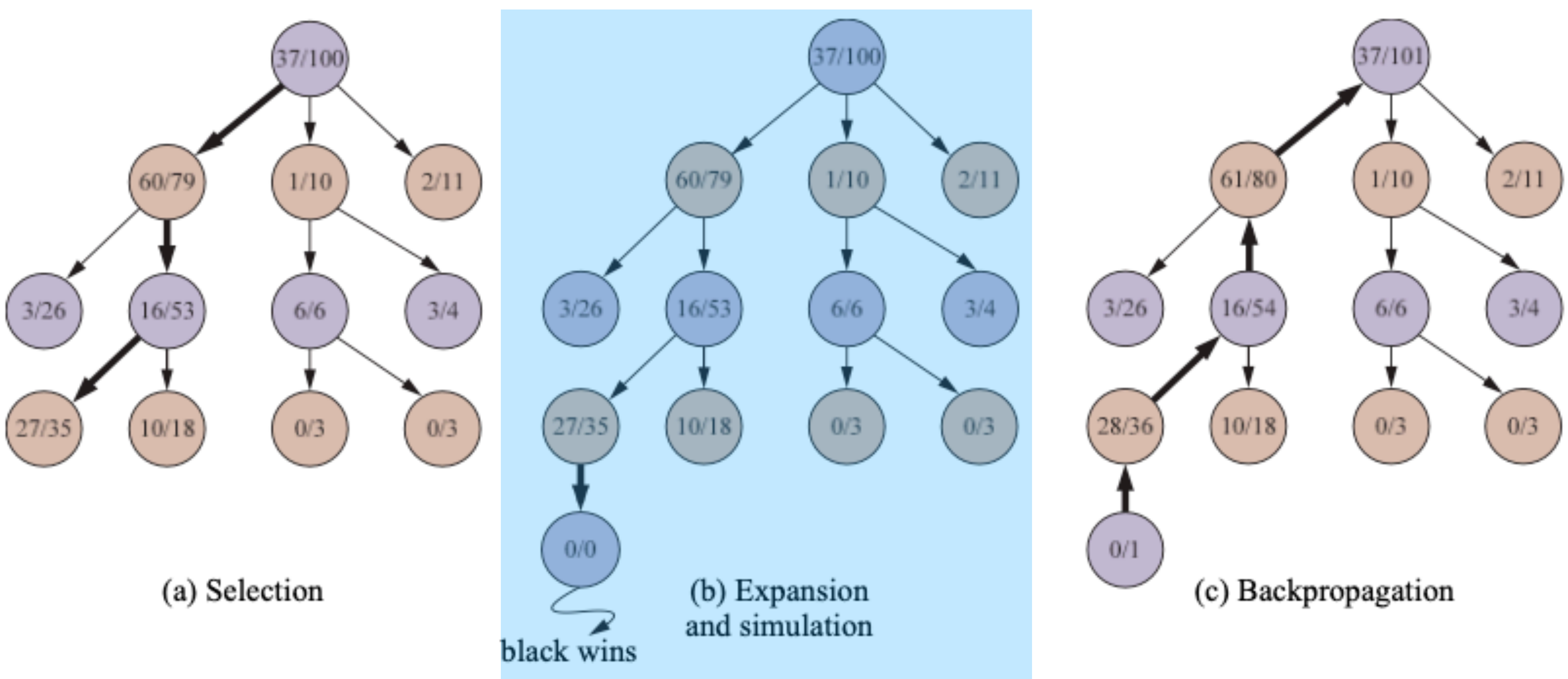
(a) Selection
(b) Expansion and simulation
(c) Backpropagation

S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

モンテカルロ木探索（MCTS, Monte-Carlo Tree Search）

expansion :
探索木を新規子ノードを追加して成長させる← 葉ノードの箇所に新規子ノード生成 …0/0で示されている

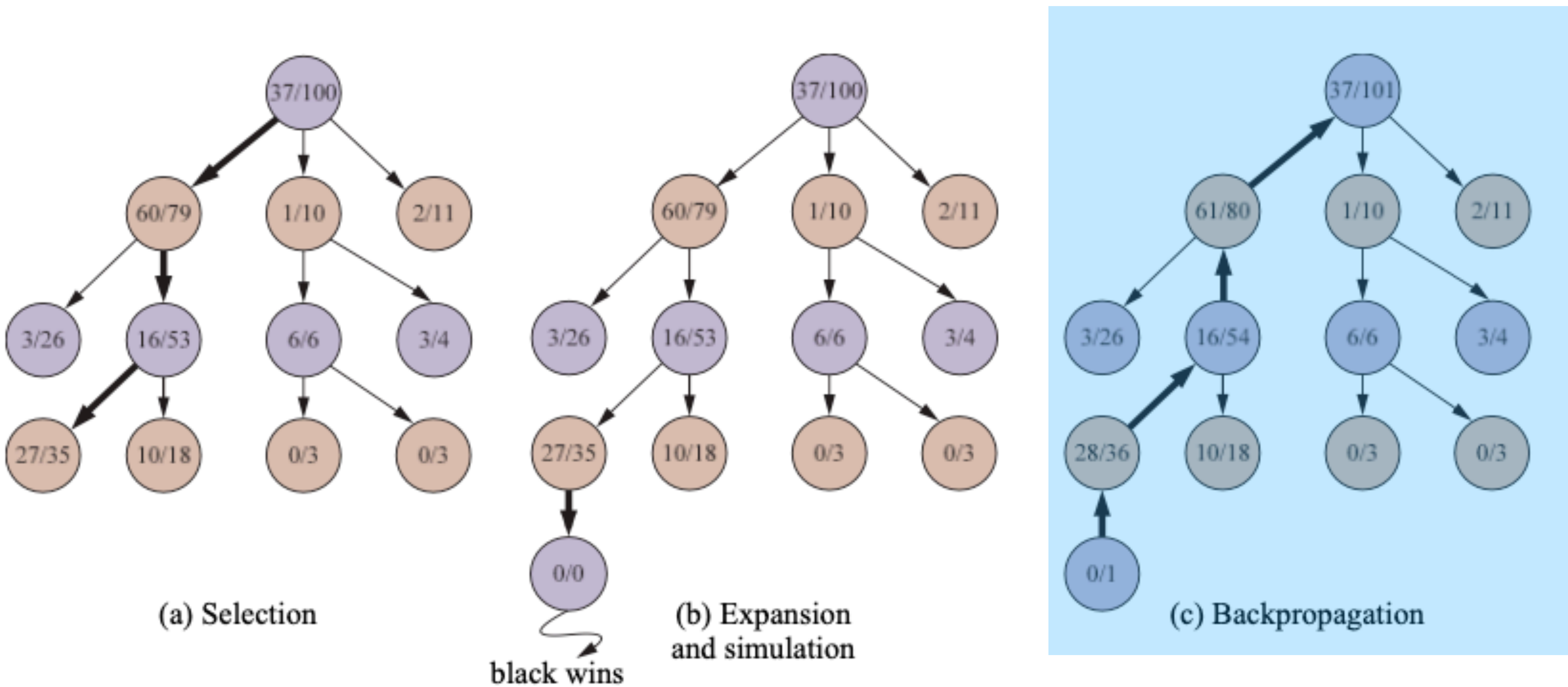
simulation :
新たに生成された子ノードからシミュレーション（playout）開始
… シミュレーションの結果blackが勝ったとする



S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

モンテカルロ木探索（MCTS, Monte-Carlo Tree Search）

back-propagation:
新規子ノードから開始したシミュレーションの結果blackが勝利→勝率データ更新
→ ルートから新規子ノードまでの経路上においてノードの勝率更新
　blackについては試行回数と勝利数をともに　+1
　whiteについては試行回数のみ　+1



S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

モンテカルロ木探索（MCTS, Monte-Carlo Tree Search）

UCT-MCTSアルゴリズム：

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

選択方策が必要→ ex. UCT（upper confidence bounds applied to trees）

UCT : 可能な手（move）を以下の式（UCB1）に基づいてランク付け

$$\mathtt{UCB1(n)} = \frac{\mathtt{U(n)}}{\mathtt{N(n)}} + C \times \sqrt{\frac{\log \mathtt{N(PARENT(n))}}{\mathtt{N(n)}}}$$

`U(n)`：ノードnを経由する全てのpalyoutsのutility総和

`N(n)`：ノードnを経由するpaluouts数

`PARENT(n)`：探索木におけるnの親ノード

S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

モンテカルロ木探索（MCTS, Monte-Carlo Tree Search）

選択方策が必要→ ex. UCT（upper confidence bounds applied to trees）
UCT : 可能な手（move）を以下の式（UCB1）に基づいてランク付け

$$\mathtt{UCB1(n)} = \frac{\mathtt{U(n)}}{\mathtt{N(n)}} + C \times \sqrt{\frac{\log \mathtt{N(PARENT(n))}}{\mathtt{N(n)}}}$$

U(n)：ノードnを経由する全てのpalyoutsのutility総和
N(n)：ノードnを経由するpaluouts数
PARENT(n)：探索木におけるnの親ノード

UCT-MCTSアルゴリズム：

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

utility平均値最大ではなく
playouts数最大！
但し通常両者は一致することが多い

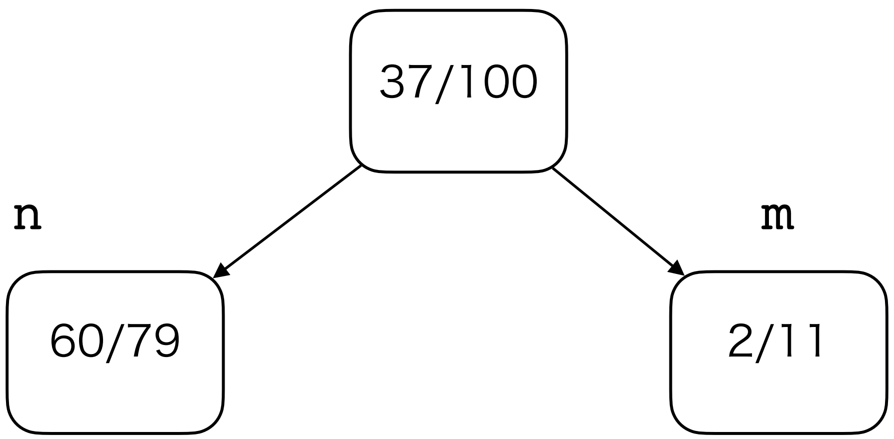
S.Russell & P.Norvig, Artificial Intelligence 4th Ed., Pearson, 2020
http://aima.cs.berkeley.edu/figures.pdf

$$\mathtt{UCB1(n)} = \frac{\mathtt{U(n)}}{\mathtt{N(n)}} + C \times \sqrt{\frac{\log \mathtt{N(PARENT(n))}}{\mathtt{N(n)}}}$$

U(n)：ノードnを経由する全てのpalyoutsのutility総和

N(n)：ノードnを経由するpaluouts数

PARENT(n)：探索木におけるnの親ノード



U(n) = 60, N(n) = 79
U(m) = 2, N(m) = 11
N(PARENT(n)) = N(PARENT(m)) = 100

UCB1(n) = 0.759 + 0.241 C, UCB1(m) = 0.182 + 0.647C

C = 1.4 ⇒ UCB1(n) = 1.10, UCB1(m) = 1.09 ⇒ n選択

C = 1.5 ⇒ UCB1(n) = 1.12, UCB1(m) = 1.15 ⇒ m選択