



# ポケモンで きょうか がくしゅう

ポケモンを れいだいとして きょうかがくしゅうの きそを みにつけよう！

なごや こうぎょう だいがく じゅきょう かみむら ともや



はじめてまして！ きょうか がくしゅうの せかいへ ようこそ！

わたしの なまえは サットウ みんなからは きかい がくしゅうの はかせと したわれて おるよ ▼

この せかいでは きょうか がくしゅう と よばれる アルゴリズム たちが

いたるところに つかわれている！ ▼

その きょうか がくしゅうを ひとつは ゲームに つかったり ロボットに つかったり・・・

そして・・・

わたしは この アルゴリズムの けんきゅうを している というわけだ ▼







いよいよ これから

きみの きょうか がくしゅうの がくしゅうの はじまりだ！ ▼

ゆめと ぼうけんと！

きょうか がくしゅうの せかいへ！

レッツ ゴー！ ▼

-  1. 問題設定
-  2. 価値関数とベルマン方程式
-  3. 最適方策を求める
-  4. 価値関数と最適方策を推定する(動的計画法)
  - 反復方策評価
  - 方策反復(Policy Iteration)
  - 価値反復(Value Iteration)
-  5. 未知環境に対する学習(モンテカルロ法)
-  6. Q学習とSARSA(TD法)



# 1. 問題設定


# ピカチュウ対コイキング

5

- ピカチュウとコイキングの戦い
- 問題はゲーム実機よりも簡単しておく
  - 技は(技に固有の)確率でヒットし, 固定値のダメージを与える



コイキング

HP:   
21/21

HP:21

持ち物なし

わざ:

はねる ダメージを与えることができない

- 攻撃し合うとややこしいので, コイキングは一切反撃できないように技を設定する
- PP切れにはならないので, わるあがきは出せない

わざ:

はねる      めいちゅう    〇    ダメージ    〇



ピカチュウ

HP:   
100/100

HP:(今回関係ない)

持ち物なし

わざ:

たいあたり 命中率100%, ダメージ5



でんじほう 命中率50%, ダメージ20

- ダメージ期待値はでんじほうの方が高い

わざ:

たいあたり   めいちゅう    100    ダメージ    5

でんじほう   めいちゅう    50    ダメージ    20

- 状態 $s_t$ は,  $t$ ターン目のコイキング  の残りHP
- 行動 $a_t$ は,  $t$ ターン目のピカチュウ  の技
- ピカチュウは1ターンに1回**行動**を行い(技を使用する), コイキングの**状態**(残りHP)が確率的に変化する
- ピカチュウの攻撃の結果, コイキングの残りHPが0または負の値になったとき, 残りHPをゼロにする
  - このときを**終端状態**として, 試合が終了する

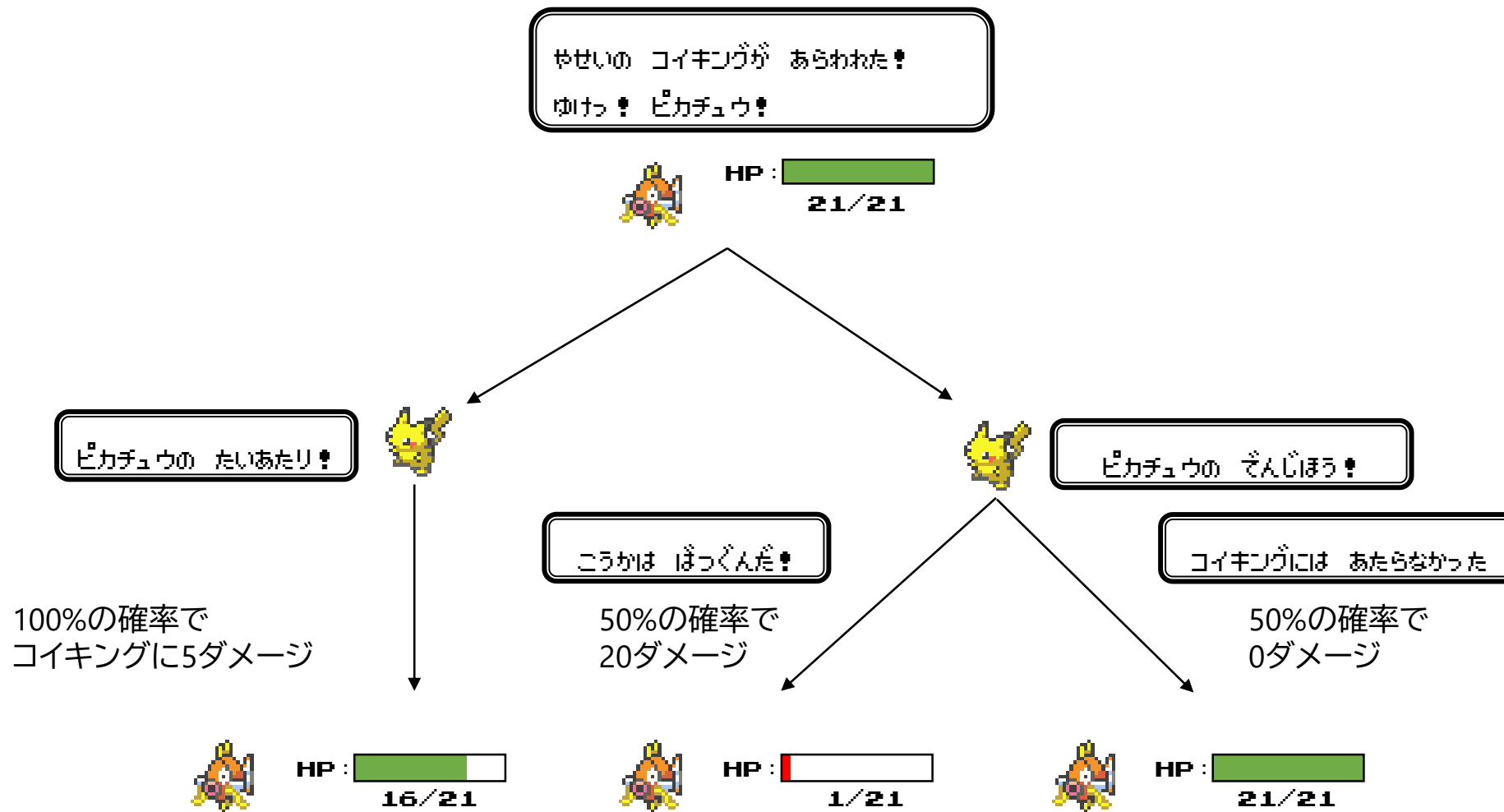
可能な状態の集合  $S = \{0, 1, 6, 11, 16, 21\}$

可能な行動の集合  $A = \{T, D\}$

T: たいあたり, D: でんじほう

# ある状態遷移(1ターン目)

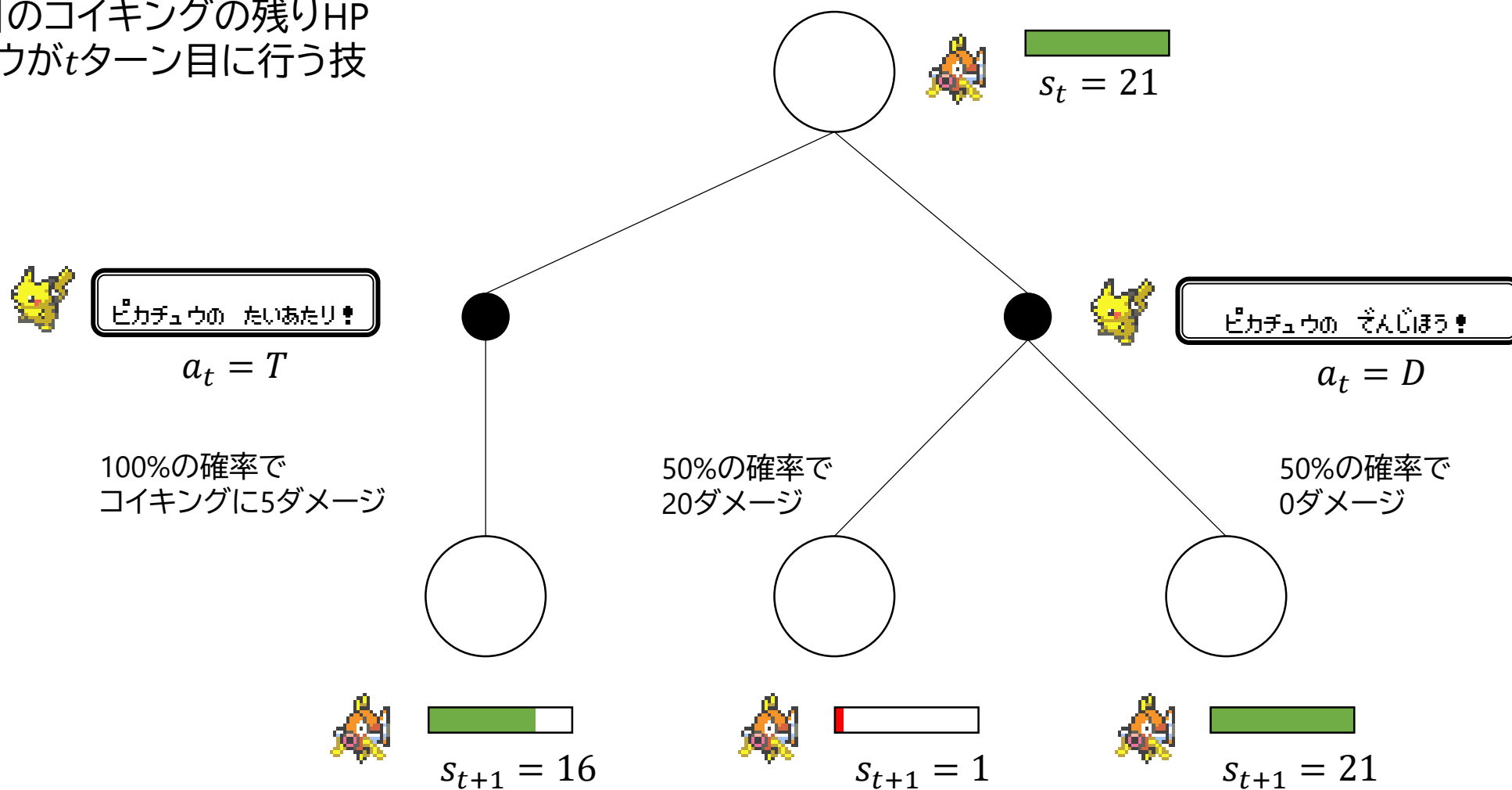
7



2ターン目のコイキングの残りHPは, ピカチュウの行動に基づいて確率的に決定する

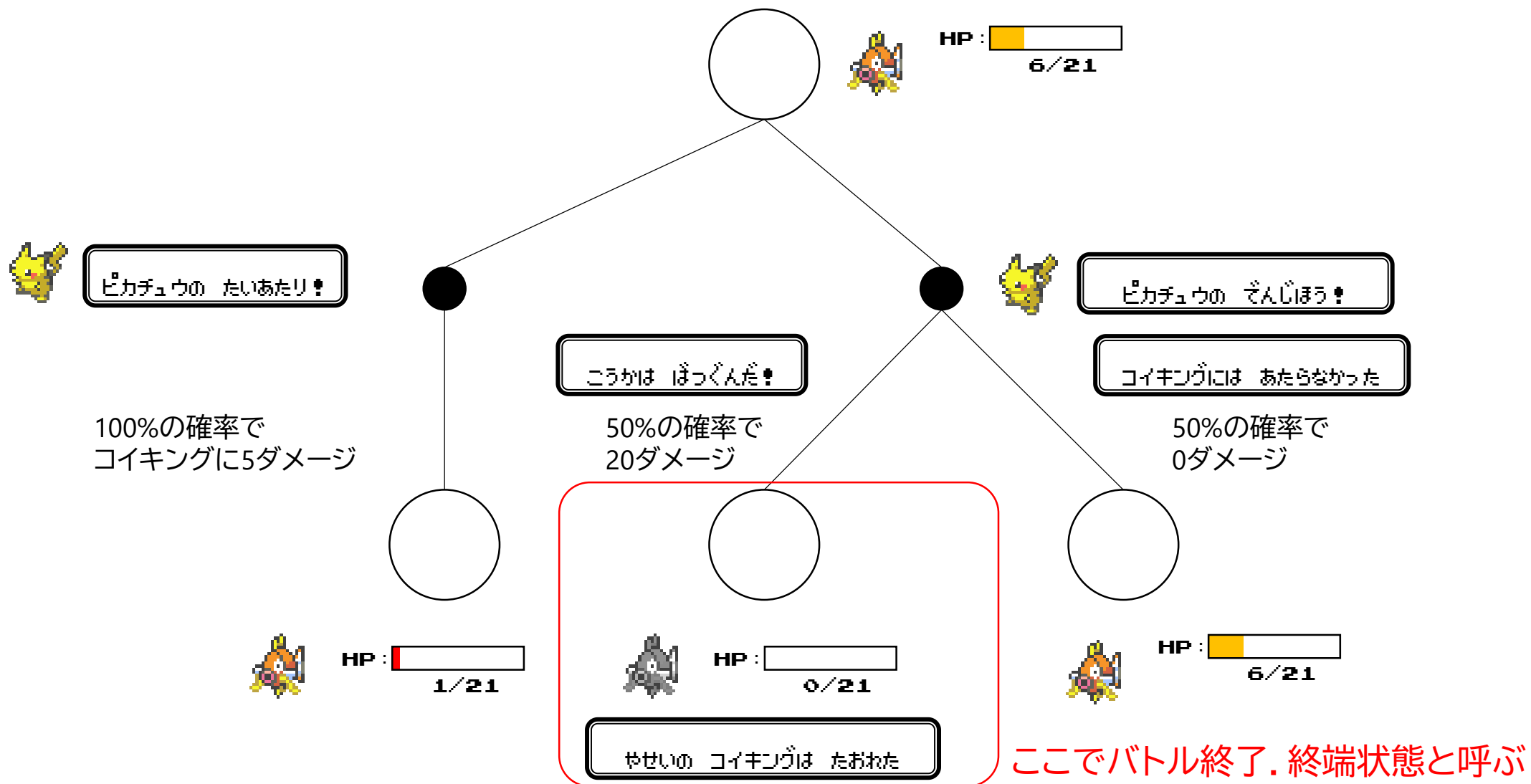
# バックアップ線図

- 状態を○, 行動を●で表す
- 状態 $s_t$ は $t$ ターン目のコイキングの残りHP
- 行動 $a_t$ はピカチュウが $t$ ターン目に行う技





# ある状態遷移(Nターン目)

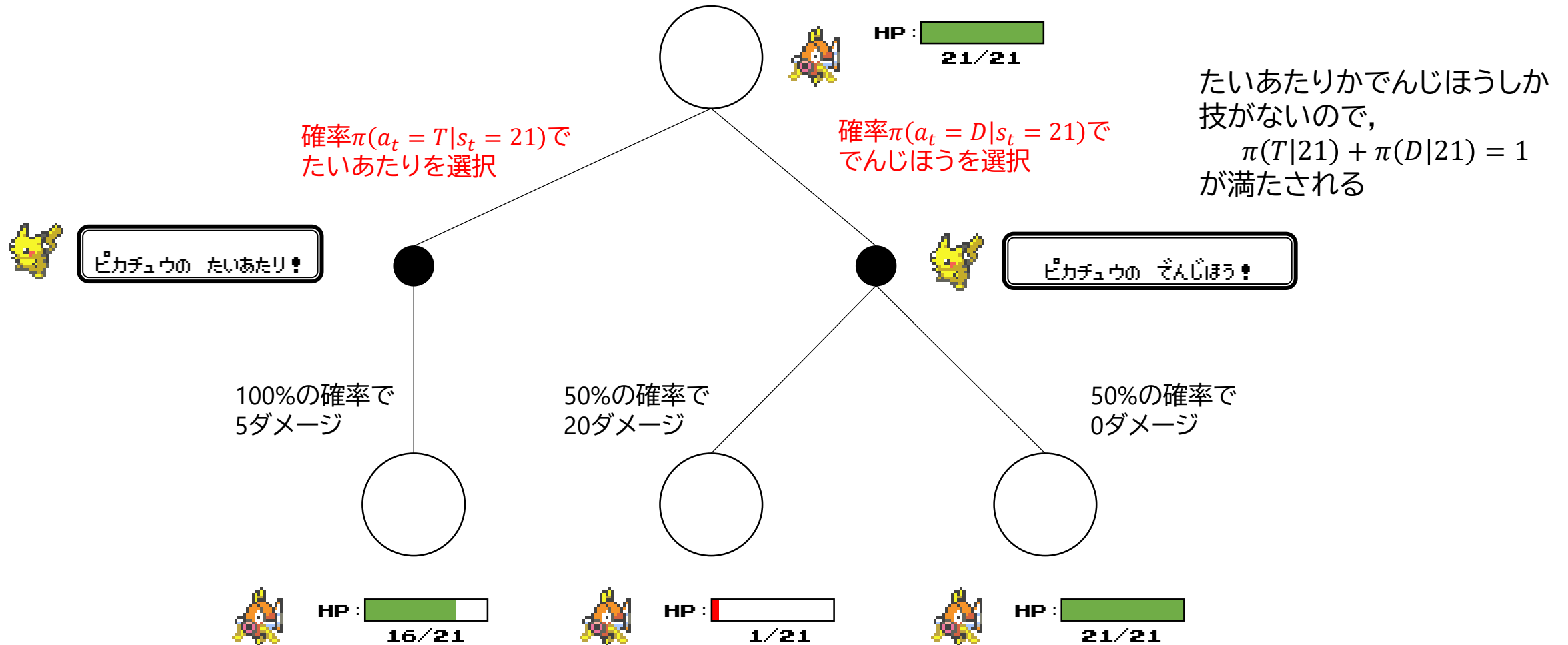


- 新しい状態 $s_{t+1}$ が, その直前の状態 $s_t$ と行動 $a_t$ のみに依存して(確率的に)決定するとき, そのような遷移を**マルコフ決定過程(Marcov decision process, MDP)**と呼ぶ
- 今回設定した問題において, コイキングの残りHPは, その直前のターンの残りHPとピカチュウの技によって確率的に決定するので, マルコフ決定過程である
  - わざのレパートリーによっては, MDPにならないこともある
  - 例えば, 次のターンの技の威力を2倍にする「じゅうでん」という技を使うと, 状態がその直前のターンだけで決定しなくなってしまう

- 以上の問題設定で, 「ピカチュウが技の選択する指針」を考える
- ここでの「技の選択指針」を, **方策** $\pi$ と呼ぶ
- $\pi(a_t|s_t)$ は現在の状態 $s_t$ に対して行動 $a_t$ を選択する確率を表す
- 一定のアルゴリズム下で, **最適方策** $\pi^*$ を導きたい
  - 最適とは, 最小のターン数でコイキングに勝利すること, と定義する

# 方策 $\pi$ に基づく行動の選択

12

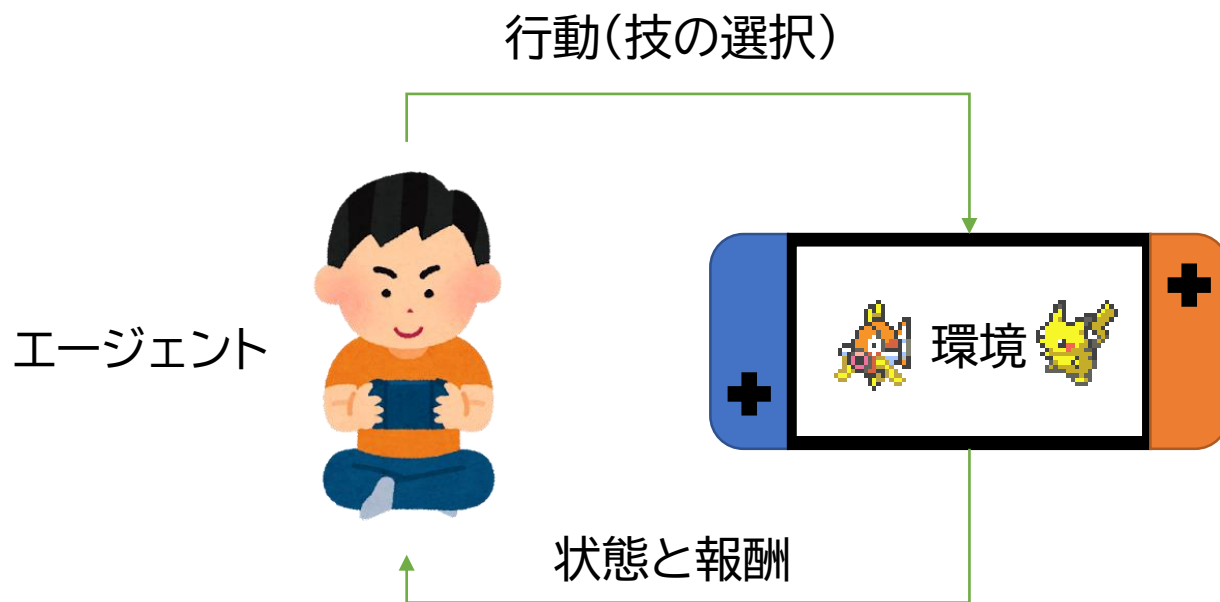


- 強化学習問題では, **エージェント**が行動を選択し, **環境**が変化する
- 環境からは, 行動の結果として新しい状態と報酬を得る

問 上記のポケモン問題において, エージェントと環境はそれぞれ何?

誤答 エージェントが「ピカチュウ」で, 環境が「コイキングとの戦闘」

正解 エージェントはゲーム「ポケモン」のプレイヤーで, 環境はポケモン世界  
(ピカチュウもコイキングも技も全て環境に含まれる)



## 間違いやすいポイント

エージェントは意思決定を行う存在  
技を繰り出しているのはピカチュウだが,  
その**技を選択したのはプレイヤー**なので  
エージェント=プレイヤーである

## 2. 価値関数とベルマン方程式

- 方策の良し悪しを評価するために, **報酬**  $r_t$  を設定する
- コイキングに勝利することが目的なので,  $s_{t+1} = 0$  となった瞬間に  $r_t = 10$  を与える
- 長々と戦うことに価値はないので, それ以外の状態では  $r_t = -1$  を与える
- 戦いが終わったあとは何をしても変わらないので,  $r_t = 0$  を与える
- 累計の報酬

$$G_t = \sum_{i=1}^t r_i = r_1 + r_2 + \cdots + r_t$$

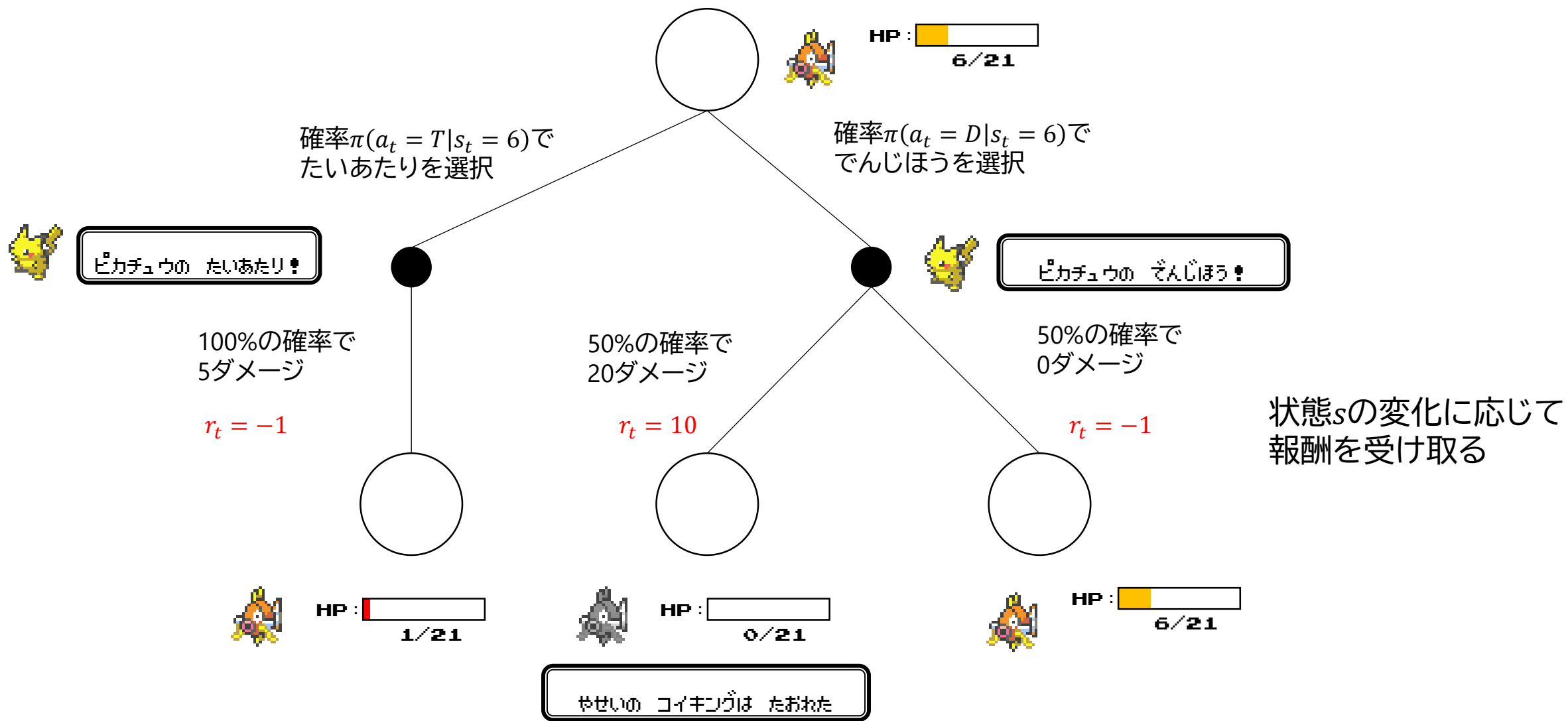
を求める. これが方策の良し悪しを判断する基準になる

- 即時的な報酬ではなく、累計報酬を基準とすることが強化学習の特徴
- 即時的な報酬だけを最大化するならば、  
ダメージ期待値の大きい「でんじほう」を常に選択すればよいだろう
- しかし、実際にそのような戦略が最適ではないことは明らか



# 報酬が得られる様子

17



- 終端状態がないような問題を考えるとき, 累計報酬は発散してしまう
- 仮に終端状態があったときにも, はじめのうちに大きな報酬を得る場合と, ものすごく時間が立ってから大きな報酬を得る場合とでは, 前者のほうが良いはず
- このような問題に対処するため, **割引率** $\gamma$ を設定することがある

$$G_t = \sum_{i=1}^{\infty} \gamma^i r_i = r_1 + \gamma r_2 + \cdots + \gamma^n r_n + \cdots$$

- **状態価値関数**  $v_\pi(s)$  は, 状態  $s$  から始めて, 方策  $\pi$  に従ったとき, その後全部の報酬の期待値を表す.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right]$$

- 状態価値関数  $v_\pi(s_t)$  は, 次の状態価値関数  $v_\pi(s_{t+1})$  を用いて以下のように表せる. これを **ベルマン方程式** と呼ぶ

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

$$= \mathbb{E}_\pi \left[ R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s \right]$$

$$= \sum_a \pi(a|s) \left\{ \sum_{s', r} p(s', r | s, a) \{ r + \gamma v_\pi(s') \} \right\}$$

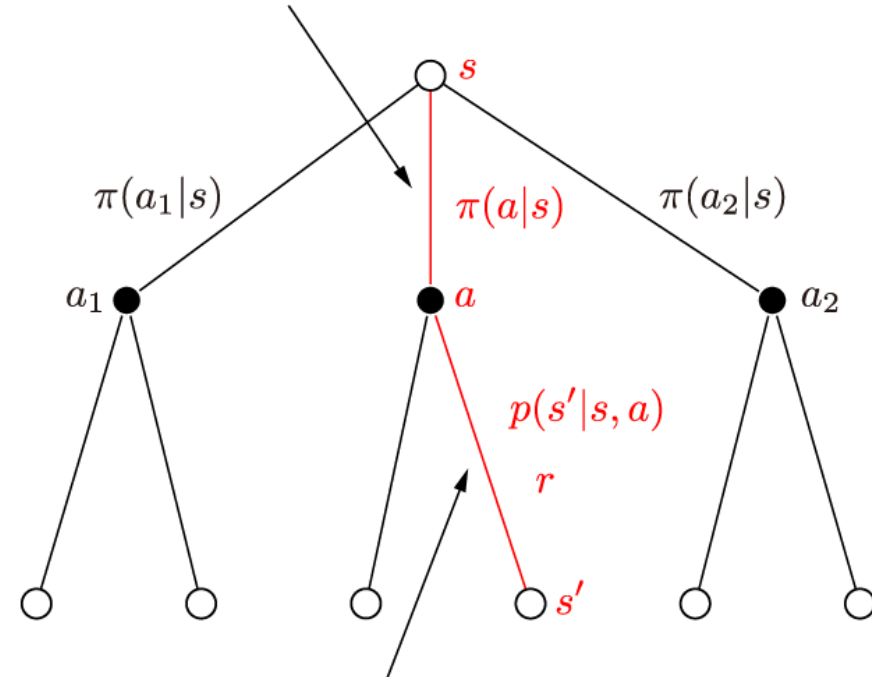
$a$  を行う確率

$s'$  に遷移する確率

$s'$  に遷移した  
ことで  
得られる報酬

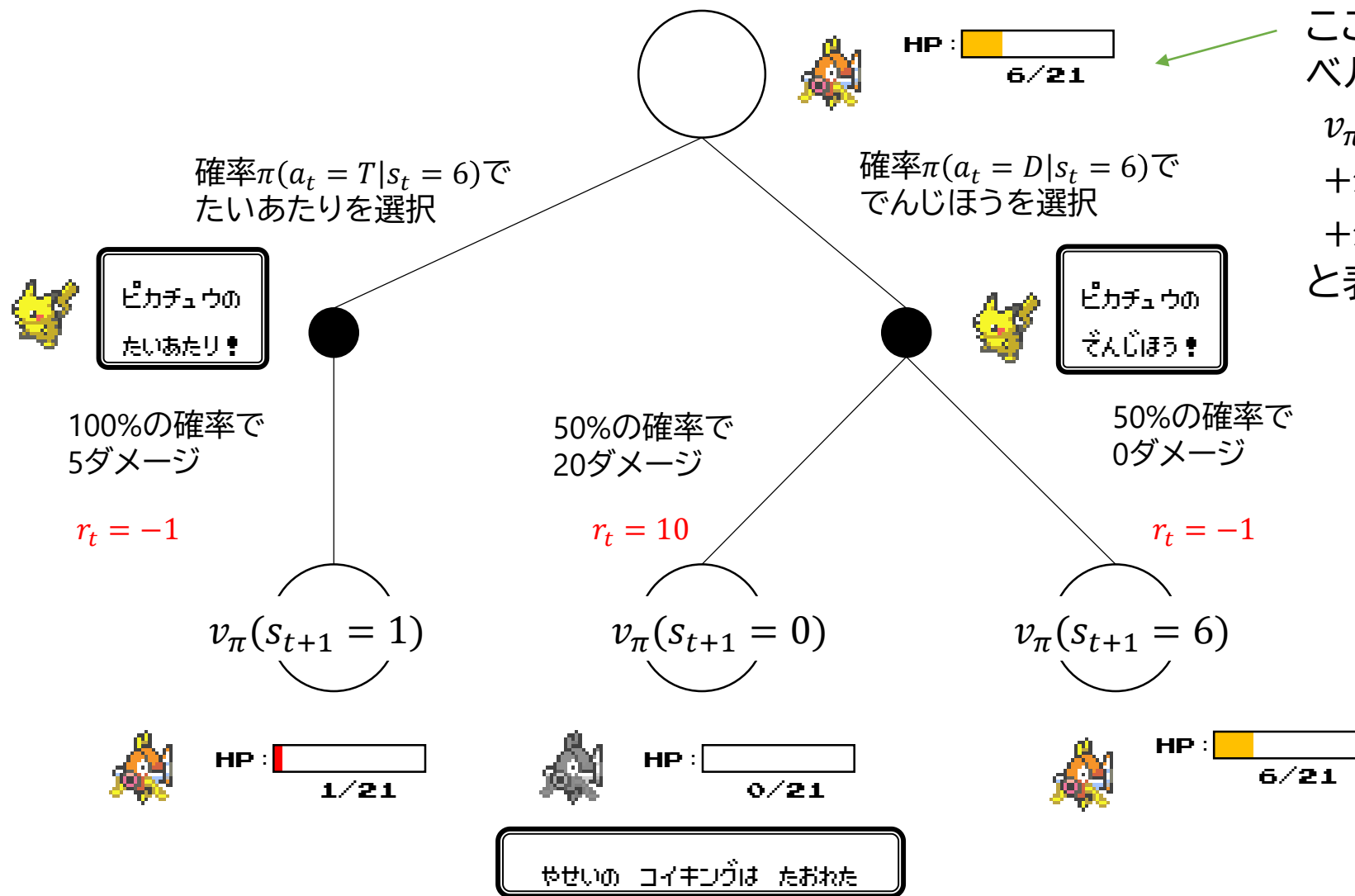
今後得られる報酬  
(割引含む)

① 状態  $s$  でポリシー  $\pi(a|s)$  に従って確率的に行動  $a$  を選択



② 行動  $a$  を行うと, 確率  $p(s'|s, a)$  で状態  $s'$  に遷移  
そのとき報酬  $r$  を得る

# ある場合のベルマン方程式



ここでの価値 $v_\pi(s_t = 6)$ は  
ベルマン方程式にもとづき

$$\begin{aligned} v_\pi(6) = & \pi(T|6) \times 1 \times (-1 + \gamma v_\pi(1)) \\ & + \pi(D|6) \times 0.5 \times (10 + \gamma v_\pi(0)) \\ & + \pi(D|6) \times 0.5 \times (-1 + \gamma v_\pi(6)) \end{aligned}$$

と表すことができる

# ベルマン方程式を解いてみる①

22

簡単な場合には、ベルマン方程式を解析的に解くことができる

例1:  $\pi(T) = 1, \pi(D) = 0$  の場合. すなわち, たいあたりしかない場合.  $\gamma = 1$  として割引なし.



まずはそれぞれの状態に対してベルマン方程式を書き下してみる

$$v_T(s_t = 21) = 1 \times 1 \times (-1 + v_T(s_{t+1} = 16)) = v_T(16) - 1$$

行動選択確率

状態遷移確率

同様に  $v_T(16) = v_T(11) - 1$

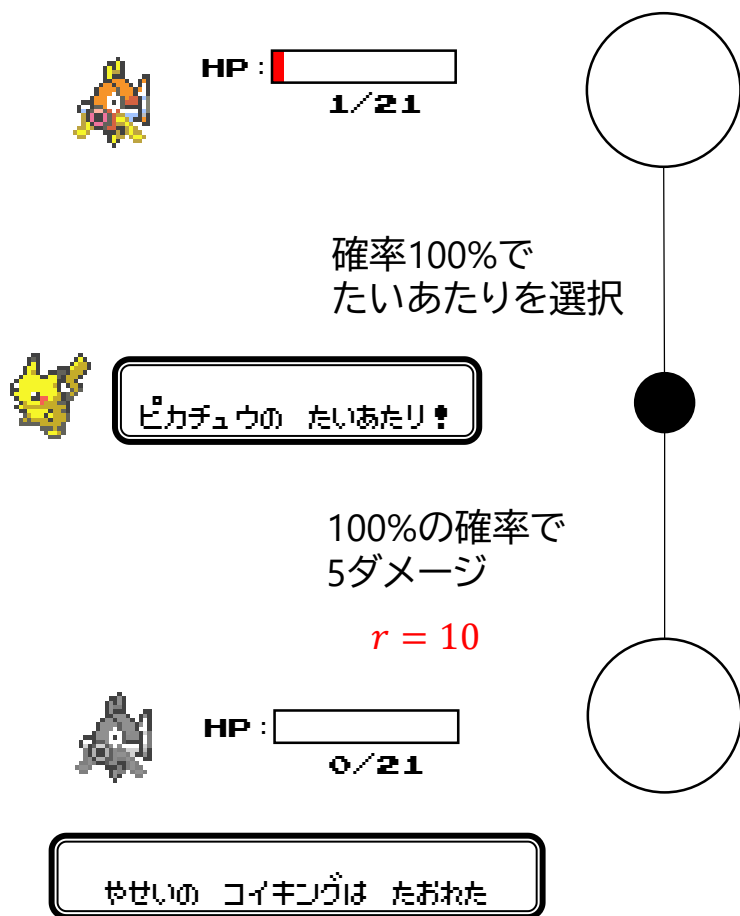
$$v_T(11) = v_T(6) - 1$$

$$v_T(6) = v_T(1) - 1$$

# ベルマン方程式を解いてみる①

23

例1:  $\pi(T) = 1, \pi(D) = 0$  の場合. すなわち, 「たいあたり」しかない場合.  $\gamma = 1$  として割引なし.



終端状態では大きな報酬がもらえる

$$v_T(s_t = 1) = 1 \times 1 \times (10 + v_T(s_{t+1} = 0)) = v_T(0) + 10$$

終端状態に達した以降は報酬が発生しないので,

$$v_T(0) = 0$$

あとは終端状態から逆に辿って, 具体的な数値を求めていこう

$$v_T(1) = v_T(0) + 10 = 10$$

$$v_T(6) = v_T(1) - 1 = 9$$

$$v_T(11) = v_T(6) - 1 = 8$$

$$v_T(16) = v_T(11) - 1 = 7$$

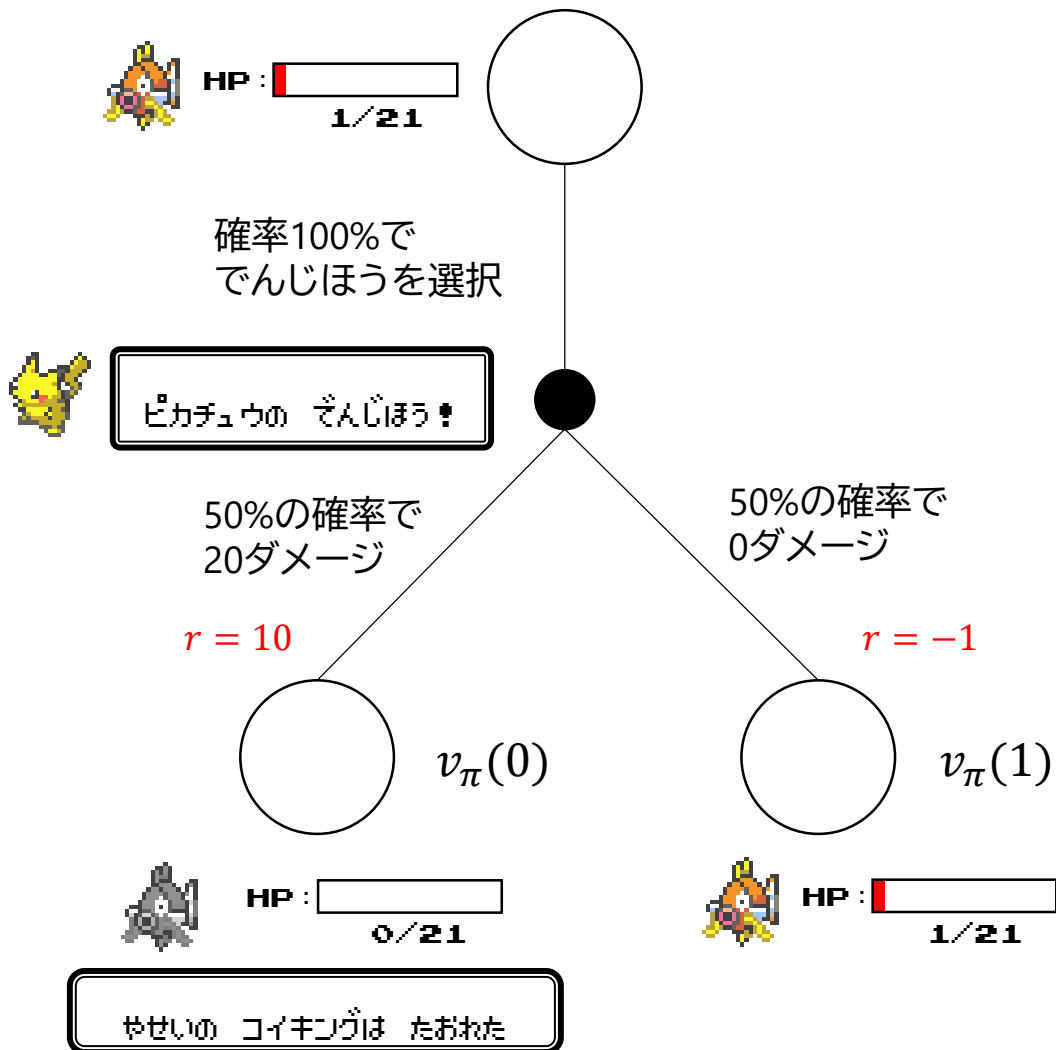
$$v_T(21) = v_T(16) - 1 = 6$$

これで方策  $\pi(T) = 1$  に対してすべての状態におけるベルマン方程式が解けた

# ベルマン方程式を解いてみる②

24

例2:  $\pi(T) = 0, \pi(D) = 1$  の場合. すなわち, 「でんじほう」しかない場合.  $\gamma = 1$  として割引なし.



例1から, 終端状態から逆に辿っていくのがよい  
終端状態では価値関数は

$$v_D(s_t = 0) = 0$$

次に, 残りHPが1のとき(左のバックアップ線図)

$$v_D(s_t = 1) = 1 \times 0.5 \times (10 + v_D(0)) + 1 \times 0.5 \times (-1 + v_D(1))$$

これを整理して解くと

$$v_D(1) = 9$$

を得る

同様に

$$v_D(6) = v_D(11) = v_D(16) = 9$$

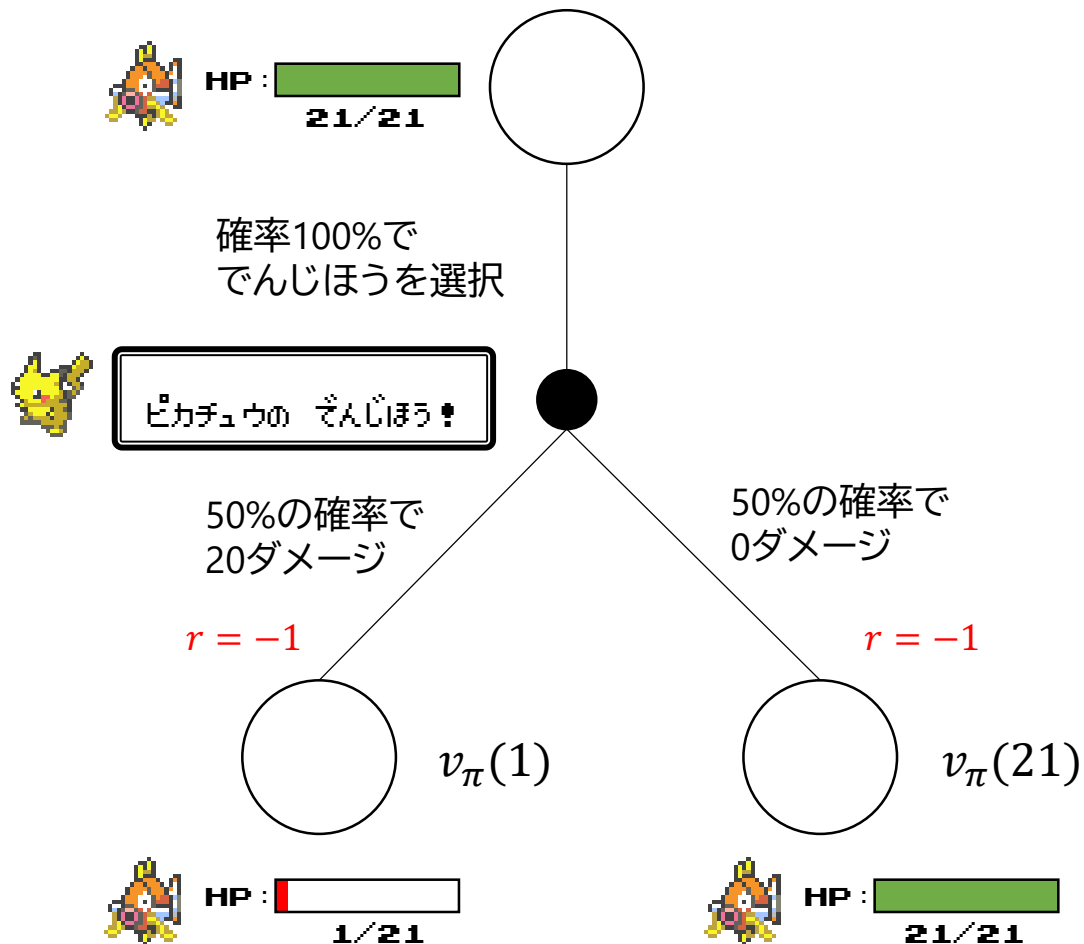
である



# ベルマン方程式を解いてみる②

25

例2:  $\pi(T) = 0, \pi(D) = 1$ の場合. すなわち, 「でんじほう」しかない場合.  $\gamma = 1$ として割引なし.



HPが満タンから始まったとき(左のバックアップ線図)

$$v_D(s_t = 21) = 1 \times 0.5 \times (-1 + v_D(1)) + 1 \times 0.5 \times (-1 + v_D(21))$$

これを整理して解くと

$$v_D(21) = 7$$

これで方策 $\pi(D) = 1$ に対して.  
すべての状態におけるベルマン方程式が解けた.

- たいあたりしかしない場合と、でんじほうしかしない場合の状態価値関数を比較してみる

$s$	$v_T(s)$	$v_D(s)$
21	6	7
16	7	9
11	8	9
6	9	9
1	10	9
0	0	0

← 基本的には、でんじほうだけを選ぶ方策の方が価値関数が高い

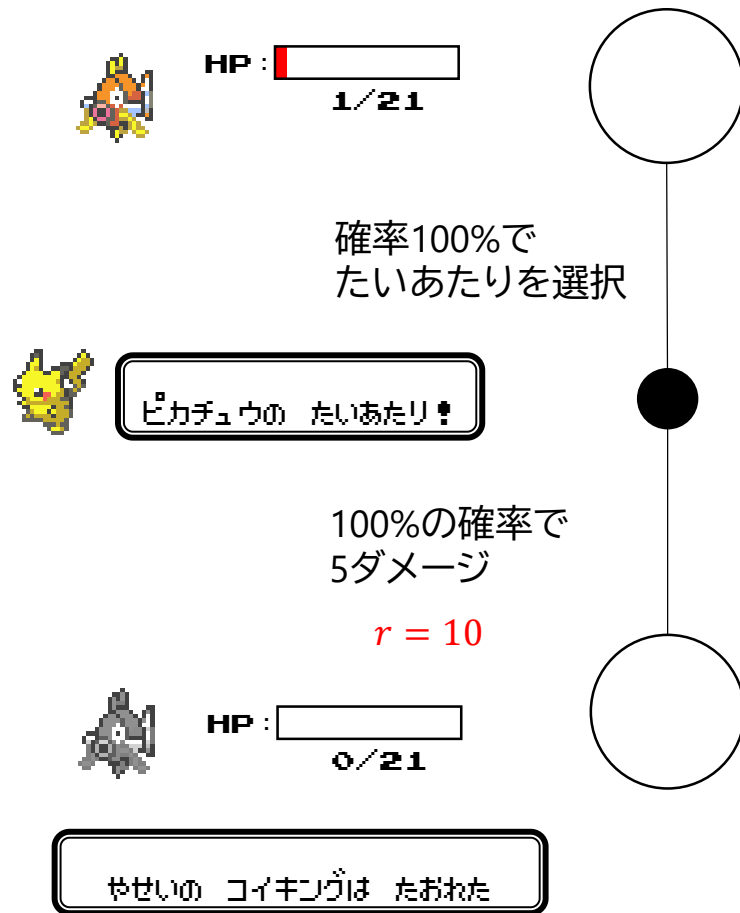
← しかし、残りHPが1のときには価値が逆転している  
当然予想される通り、「でんじほうだけを選ぶ」という方策は最適ではなさそうだ

以上の結果から、  
「残りHPが1のときには必ずたいあたりを選択し、他の場合は必ずでんじほうを選択」  
という方策が最適(最も効率よくコイキングを倒せる)であることが予測できる

# 改善した方策 $\pi^*$ のベルマン方程式を解く

27

改善した方策: 残りHPが1のときにはたいあたりを, 他の場合にはでんじほうを選択する



この方策に対して状態価値関数を求めていく

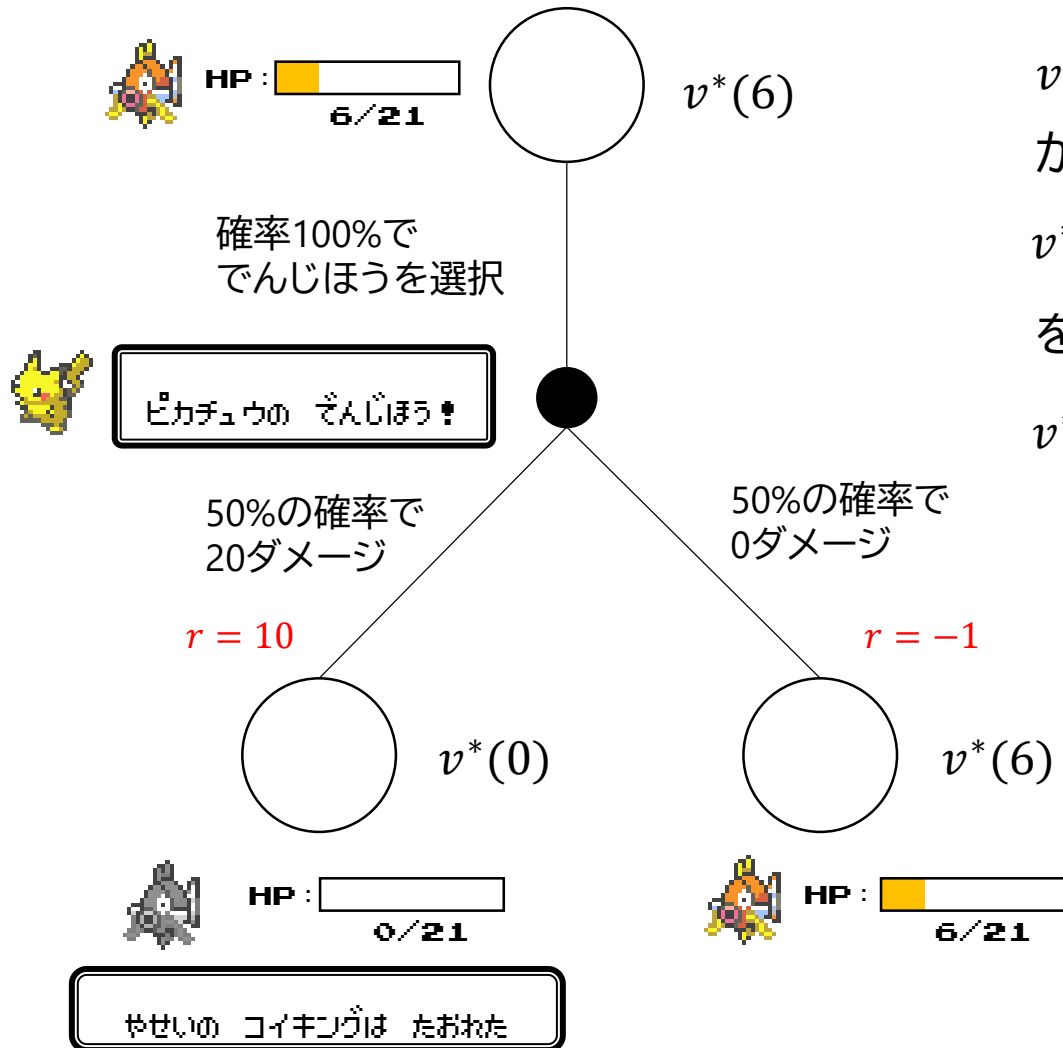
$$v^*(s_t = 0) = 0$$

$$v^*(s_t = 1) = 1 \times 1 \times (10 + v_T(0)) = v_T(0) + 10 = 10$$

# 改善した方策 $\pi^*$ のベルマン方程式を解く

28

改善した方策:残りHPが1のときにはたいあたりを, 他の場合にはでんじほうを選択する



$$v^*(s_t = 6) = 1 \times 0.5 \times (10 + v^*(0)) + 1 \times 0.5 \times (-1 + v^*(6))$$

から

$$v^*(6) = 9$$

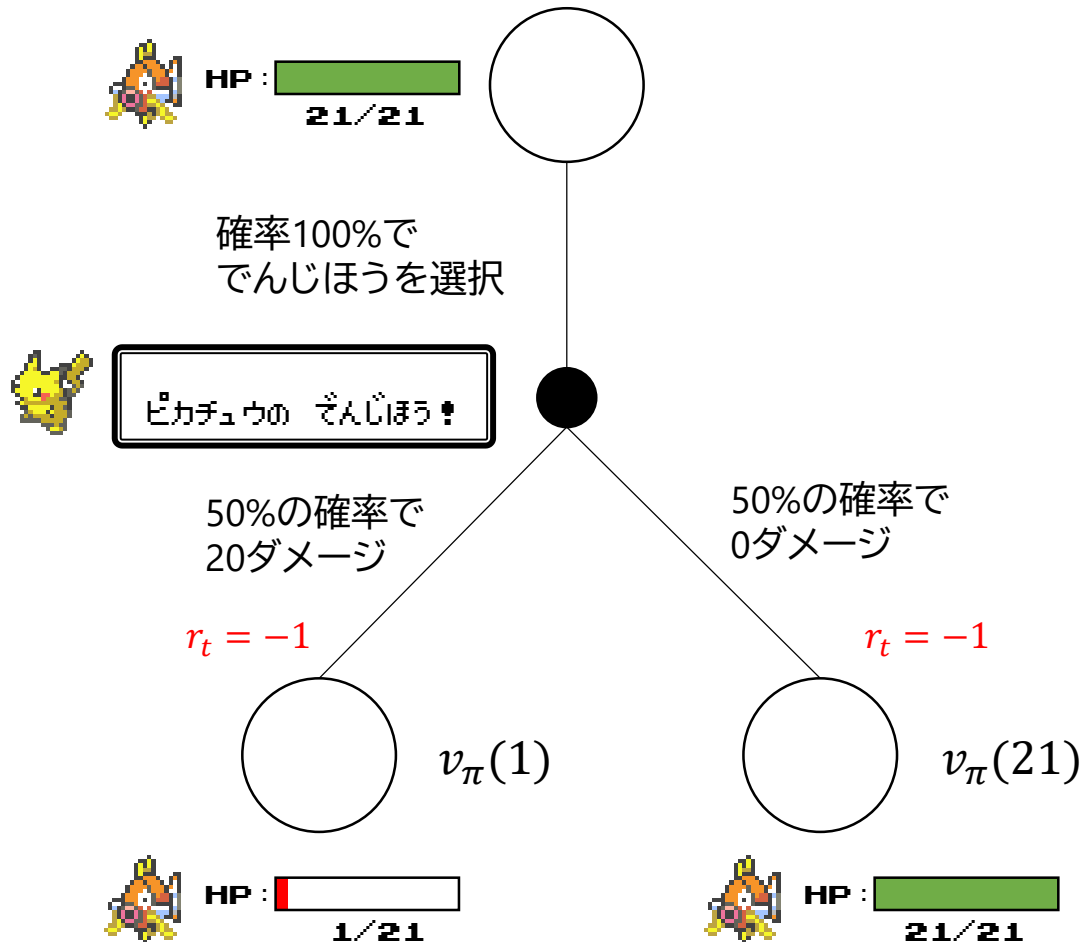
を得る. 同様にして

$$v^*(11) = v^*(16) = 9$$

# 改善した方策 $\pi^*$ のベルマン方程式を解く

29

最適方策:残りHPが1のときにはたいあたりを, 他の場合にはでんじほうを選択する



HPが満タンから始まったとき(左のバックアップ線図)

$$v^*(s_t = 21) = 1 \times 0.5 \times (-1 + v^*(1)) + 1 \times 0.5 \times (-1 + v^*(21))$$

これを整理して解くと

$$v_D(21) = 8$$

これで最適方策 $\pi^*$ に対して  
すべての状態におけるベルマン方程式が解けた

# 状態価値関数の比較(改善した方策も含む)

30

- 改善した方策に対する状態価値関数が求められたので, 他の方策における価値関数と比較してみる

$s$	$v_T(s)$	$v_D(s)$	$v^*(s)$
21	6	7	8
16	7	9	9
11	8	9	9
6	9	9	9
1	10	9	10
0	0	0	0

すべての場合において,  $v^*(s) \geq v_T(s)$  と  $v^*(s) \geq v_D(s)$  が成立している  
提案した方策は, 確かに他の方策よりも優れていることが証明できた

今後解決すべき課題: この方策は"最適"方策だろうか? 他にもっと良い方策は存在する?

### 3. 最適方策を求める

- どの行動を取るのが最適なのかを考えるための指標が欲しい
- **行動価値関数**  $q_\pi(s, a)$  は、状態  $s$  において **行動  $a$  を実行** し、その後方策  $\pi$  に従ったとき、その後全部の報酬の期待値を表す。

$$q_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]$$

状態価値関数のベルマン方程式と比較すると、  
**行動  $a$  が決定している**ところが異なる

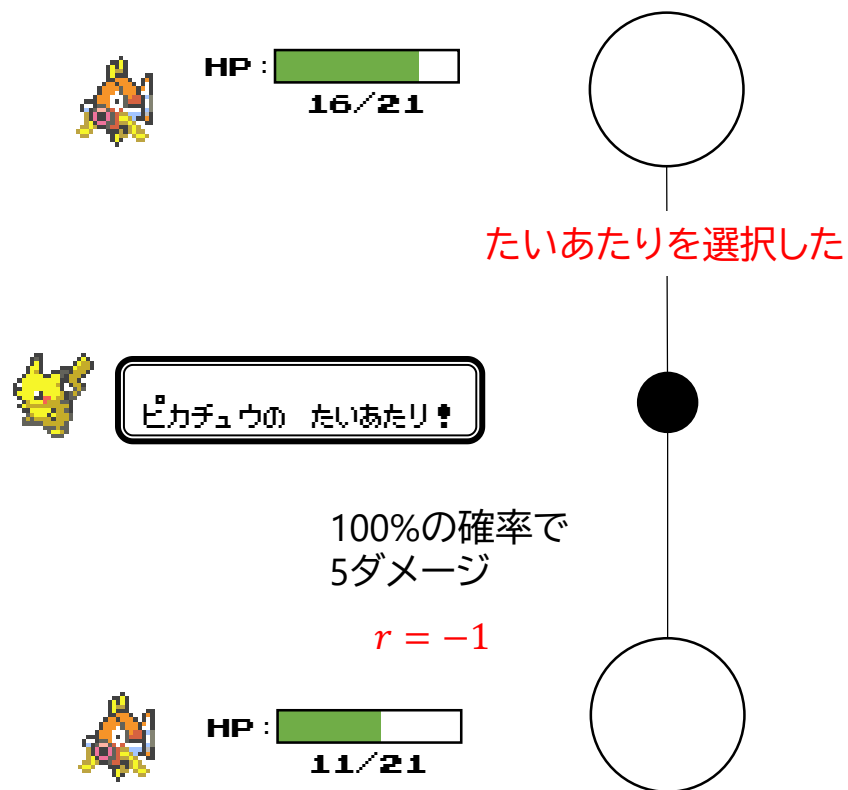
$$= \sum_{s', r} p(s', r | s, a) \{r + \gamma v_\pi(s')\}$$

$$v_\pi(s) = \sum_a \pi(a|s) \left\{ \sum_{s', r} p(s', r | s, a) \{r + \gamma v_\pi(s')\} \right\}$$



# 行動価値関数を求めてみよう

例:  $\pi(T) = 1, \pi(D) = 0$  の場合. すなわち, 「たいあたり」しかない場合.  $\gamma = 1$  として割引なし.



例えば  $s = 16$  に対して  $a = T$  を行う行動価値関数  $q_T(16, T)$  は,  $v_T(11) = 8$  であることを用いて,

$$q_T(16, T) = 1 \times (-1 + v_T(11)) = 7$$

と求められる.

他の場合についても同様に求めると以下の表のようになる.

$s$	$q_T(s, T)$
21	6
16	7
11	8
6	9
1	10
0	0

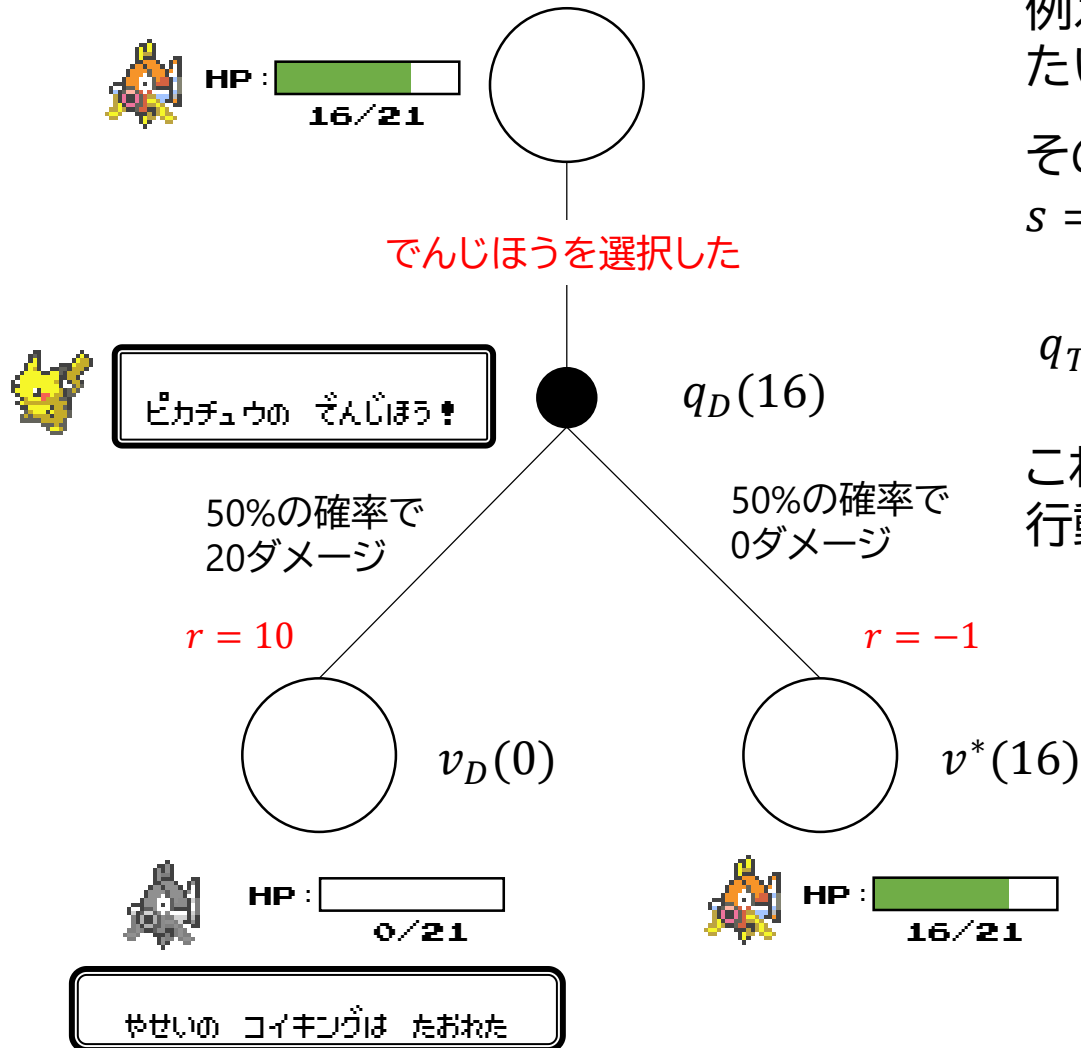
$v_T(s)$  と変わらない

- ある方策 $\pi$ が与えられたとき, ある状態 $s$ に対して, 取りうる全ての行動 $a$  (現在の方策的には選ばないものも)について行動価値関数 $q(s, a)$ を求める
- ある $a = a^*$ を取ったときの行動価値関数 $q(s, a^*)$ が, もとの方策を取り続ける場合よりも大きい値を取る場合, 方策を変更したほうがよいと判断できる

# 行動価値関数を用いて、よりよい方法を探す

35

例:  $\pi(T) = 1, \pi(D) = 0$  の場合. すなわち, たいあたりしかない場合.  $\gamma = 1$  として割引なし.



例えば  $s = 16$  の場合について,  
たいあたりではなくでんじほうを選択する.

その他の場合には元通りたいあたりのみを選択するとき,  
 $s = 16, a = D$  の行動価値関数は,  $v_T(0) = 0, v_T(16) = 7$  を用いて

$$q_{T'}(s = 16, a = D) = \frac{1}{2} \times (10 + v_T(0)) + \frac{1}{2} \times (-1 + v_T(16)) = 8$$

これは, この状況でたいあたりを選択するときの  
行動価値関数  $q_T(16, T) = 7$  と比較して

$$q_{T'}(16, D) > q_T(16, T)$$

が成立する. すなわち, この方策を改善して  
「たいあたり」よりも「でんじほう」を選択する方が  
良いことが示された.

- ある適当な初期方策 $\pi_0$ を用意して, ある状態について, それよりも優れた (すなわち, 行動価値関数が大きくなるような) 行動を選択するような, 更新方策 $\pi_1$ を見つける
- 方策 $\pi_1$ に対して, さらに優れた更新方策 $\pi_2$ を見つける... を繰り返すと, やがて最適方策 $\pi^*$ に収束することが理解できる
- このとき, 状態価値関数は最適価値関数 $v^*$ に収束する

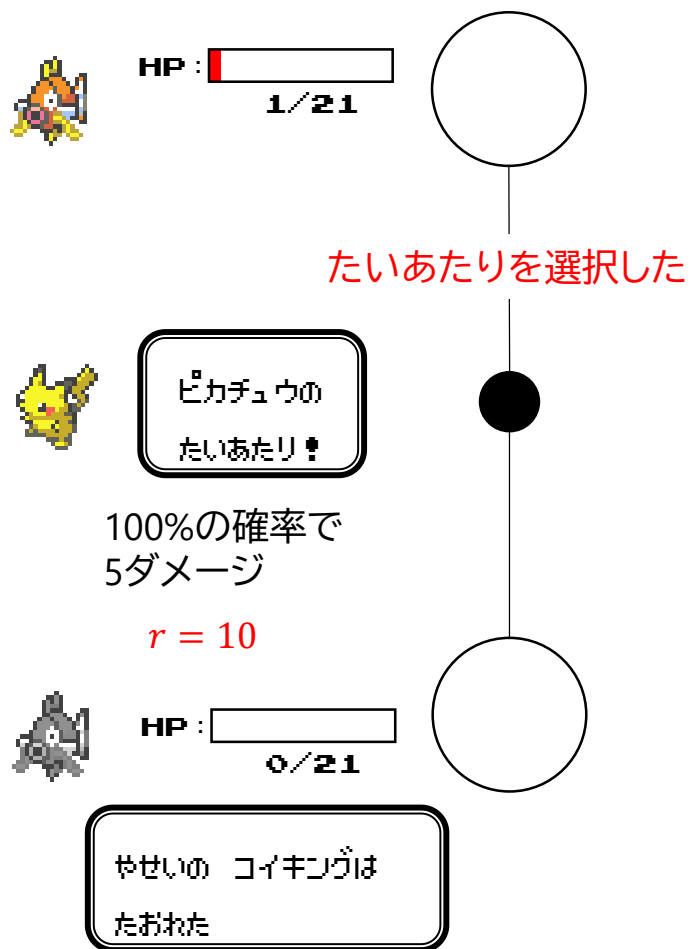
$$v^* = \max_{a \in A(s)} q_{\pi^*}(s, a)$$

# 最適方策を求めてみよう(1/5)

37

- 初期方策 $\pi_0$ を,「でんじほう」だけを選択する方策とする

ここでも, 残りHPが1のときから始める



$s = 0$ のとき, でんじほうを選択すると,

$$q(s = 1, a = D) = v_D(1) = 9$$

$s = 0$ のとき, でんじほうではなく, たいあたりを選択すると,

$$q(s = 1, a = T) = 1 \times (10 + v_D(0)) = 10$$

このとき,  $q(1, D) < q(1, T)$ が成立するので,  
 $s = 1$ のときは, でんじほうではなく, たいあたりを選択したほうが  
良いことが示された.

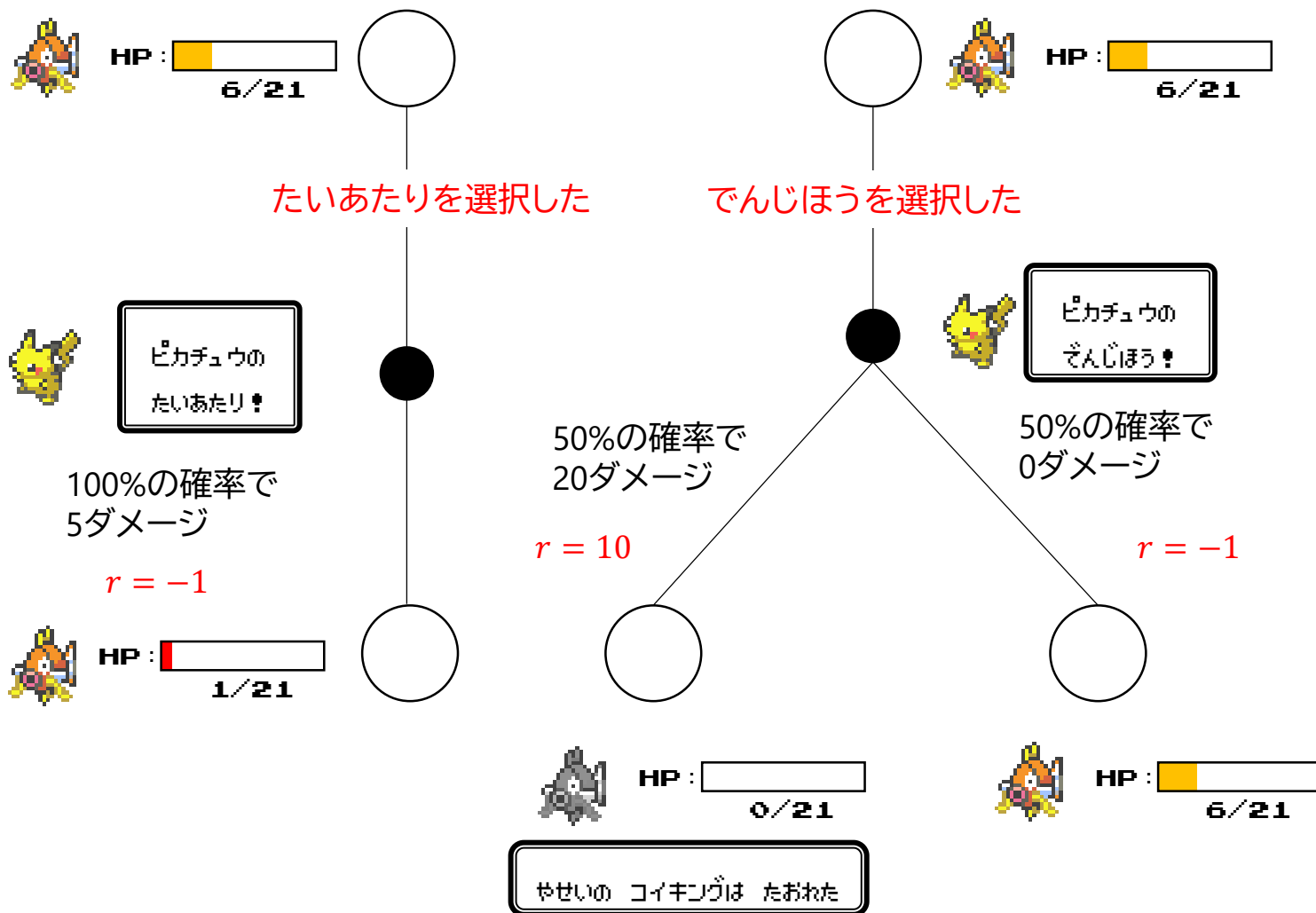
改善した方策 $\pi_1$ :

「残りHPが1のときには, 必ずたいあたりを選択する.  
他の場合には, 必ずでんじほうを選択する.」

# 最適方策を求めてみよう(2/5)

38

- 方策 $\pi_1$ をさらに更新していきたい



$s = 6$ でたいあたりを選択

$$q(6, T) = 1 \times (-1 + v_{\pi_1}(1)) = 9$$

$s = 6$ ででんじほうを選択

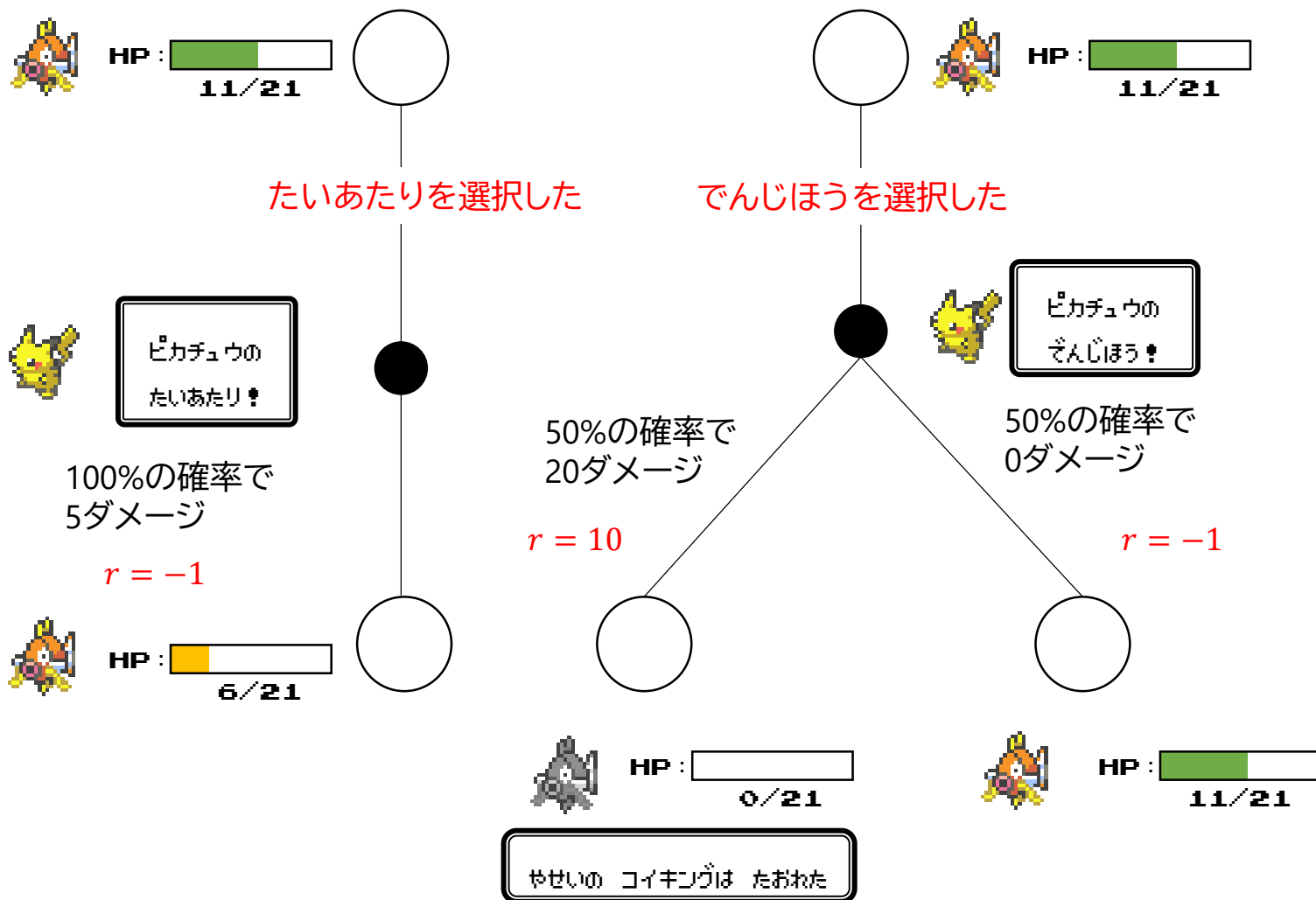
$$q(6, D) = 0.5 \times (10 + v_{\pi_1}(0)) + 0.5 \times (-1 + v_{\pi_1}(6)) = 9$$

ここでは、どちらの技を選択しても  
行動価値関数は変化しない。  
方策 $\pi_1$ をここで更新する必要はなさそうだ  
( $s = 6$ ではでんじほうを選択)

# 最適方策を求めてみよう(3/5)

39

- 方策 $\pi_1$ をさらに更新していきたい



$s = 11$ でたいあたりを選択

$$q(11, T) = 1 \times (-1 + v_{\pi_1}(6)) = 8$$

$s = 11$ ででんじほうを選択

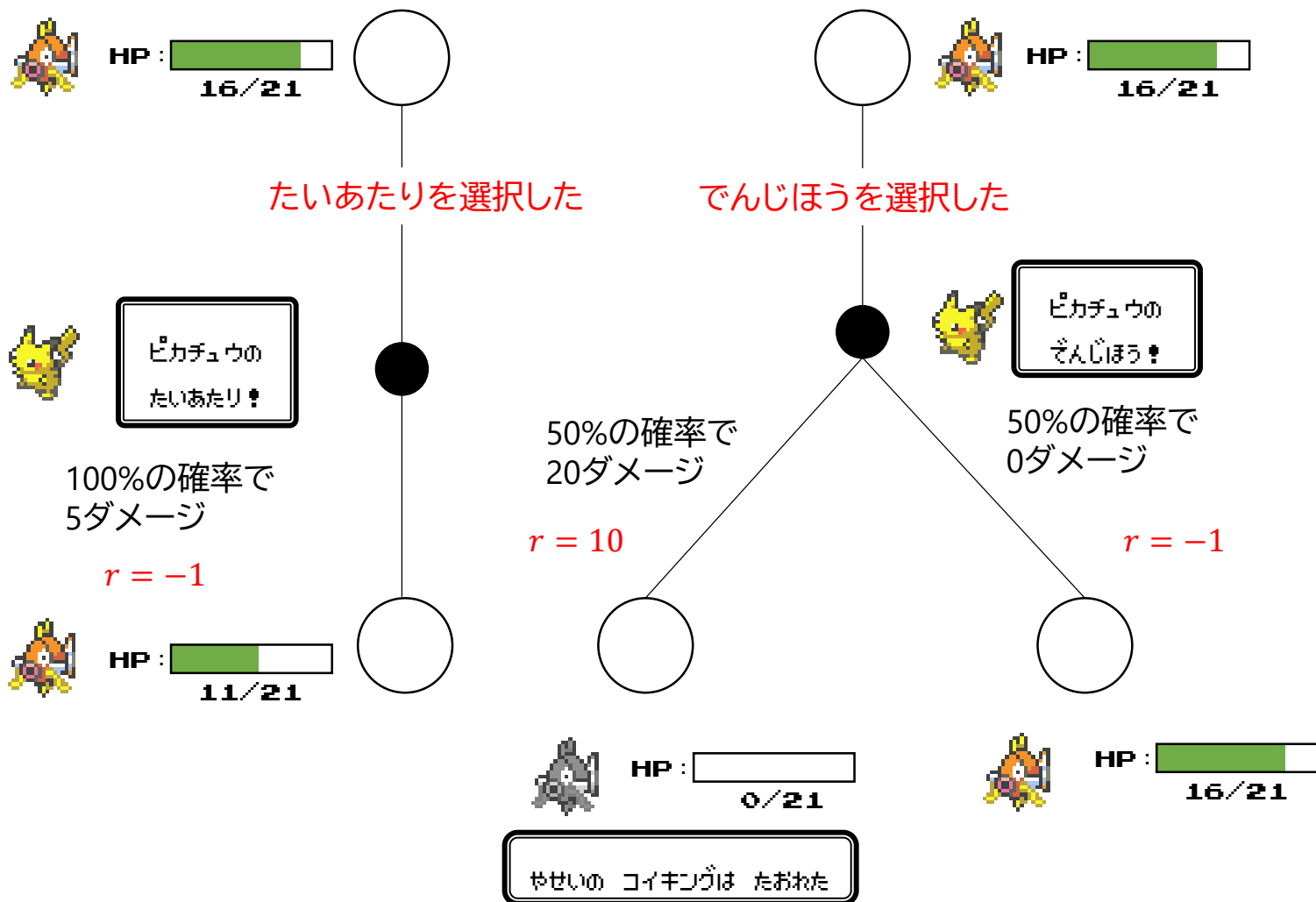
$$q(11, D) = 0.5 \times (10 + v_{\pi_1}(0)) + 0.5 \times (-1 + v_{\pi_1}(11)) = 9$$

ここでは方策 $\pi_1$ を更新する必要はない  
( $s = 11$ ではでんじほうを選択)

# 最適方策を求めてみよう(4/5)

40

- 方策 $\pi_1$ をさらに更新していきたい



$s = 16$ でたいあたりを選択

$$q(16, T) = 1 \times (-1 + v_{\pi_1}(11)) = 8$$

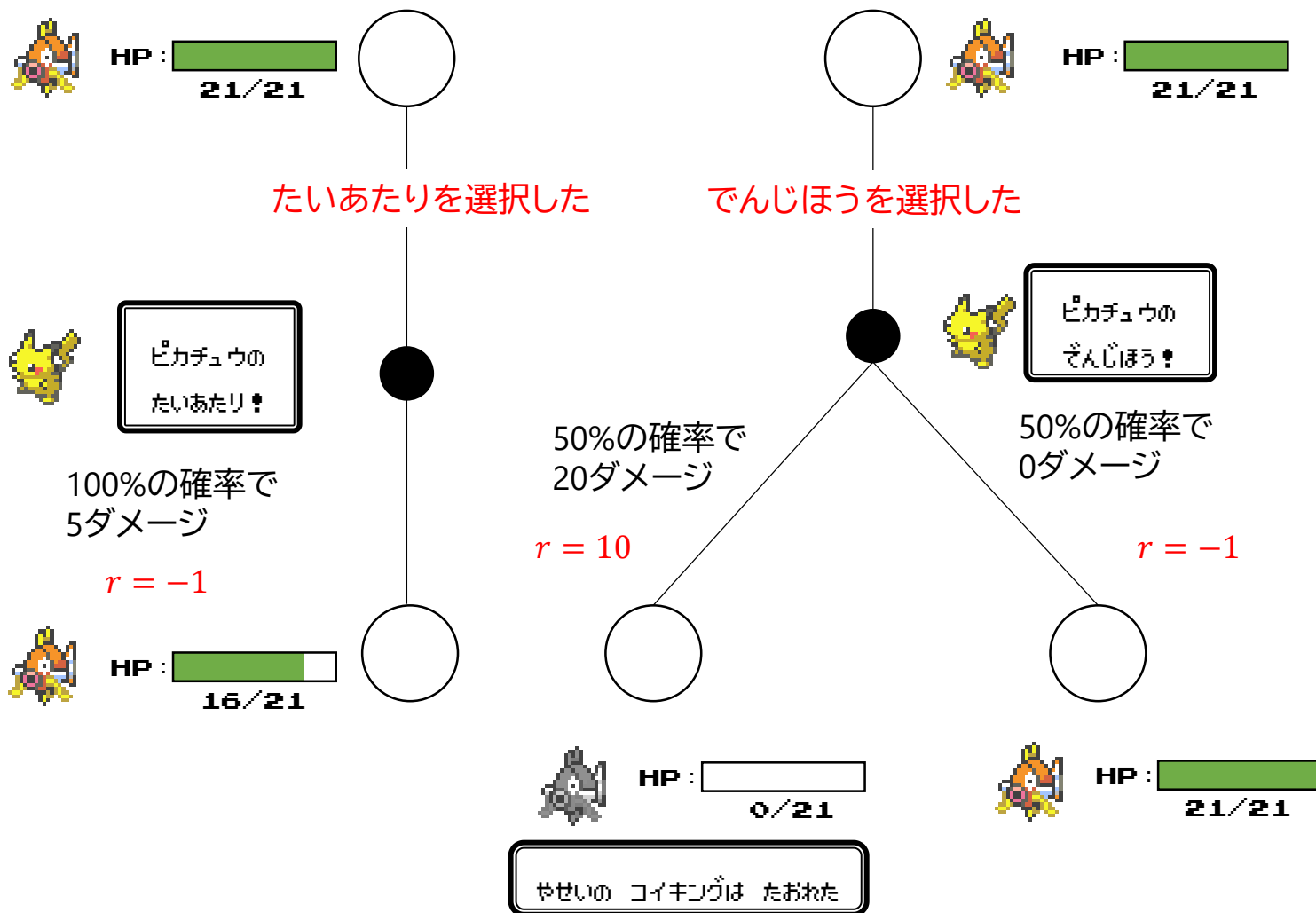
$s = 16$ ででんじほうを選択

$$q(16, D) = 0.5 \times (10 + v_{\pi_1}(0)) + 0.5 \times (-1 + v_{\pi_1}(16)) = 9$$

ここでも方策 $\pi_1$ を更新する必要はない  
( $s = 16$ ではでんじほうを選択)



- 方策 $\pi_1$ をさらに更新していきたい



$s = 21$ でたいあたりを選択

$$q(21, T) = 1 \times (-1 + v_{\pi_1}(16)) = 8$$

$s = 21$ ででんじほうを選択

$$q(21, D) = 0.5 \times (-1 + v_{\pi_1}(0)) + 0.5 \times (-1 + v_{\pi_1}(21)) = 8$$

ここでは方策 $\pi_1$ を更新してもよいし、  
しなくてもよい  
( $s = 21$ ではたいあたり, でんじほうの  
いずれを選択してもよい)

# 状態価値関数の比較(改善した方策も含む)

- 方策の更新が終了し, 最適状態価値関数 $v^*$ が求められたので, 他の方策における価値関数と比較してみる

$s$	$v_T(s)$	$v_D(s)$	$v^*(s)$
21	6	7	8
16	7	9	9
11	8	9	9
6	9	9	9
1	10	9	10
0	0	0	0

確かに $v^*(s)$ は最適方策 $\pi_1$ に対する最適価値関数になっていることが示された

- 強化学習は, 環境の**状態**に対して, エージェントが取る**行動**を決定するための**方策**を最適化する
- 方策は, 即時的な**報酬**ではなく, その累計の期待値である**価値関数**を最大化するように決定する
- **ベルマン方程式**は, 現在の状態における**状態価値関数**と次の状態における状態価値関数の関係を表す
- 方策を改善するためには, ある行動を起こしてみたときの**行動価値関数**を比較する

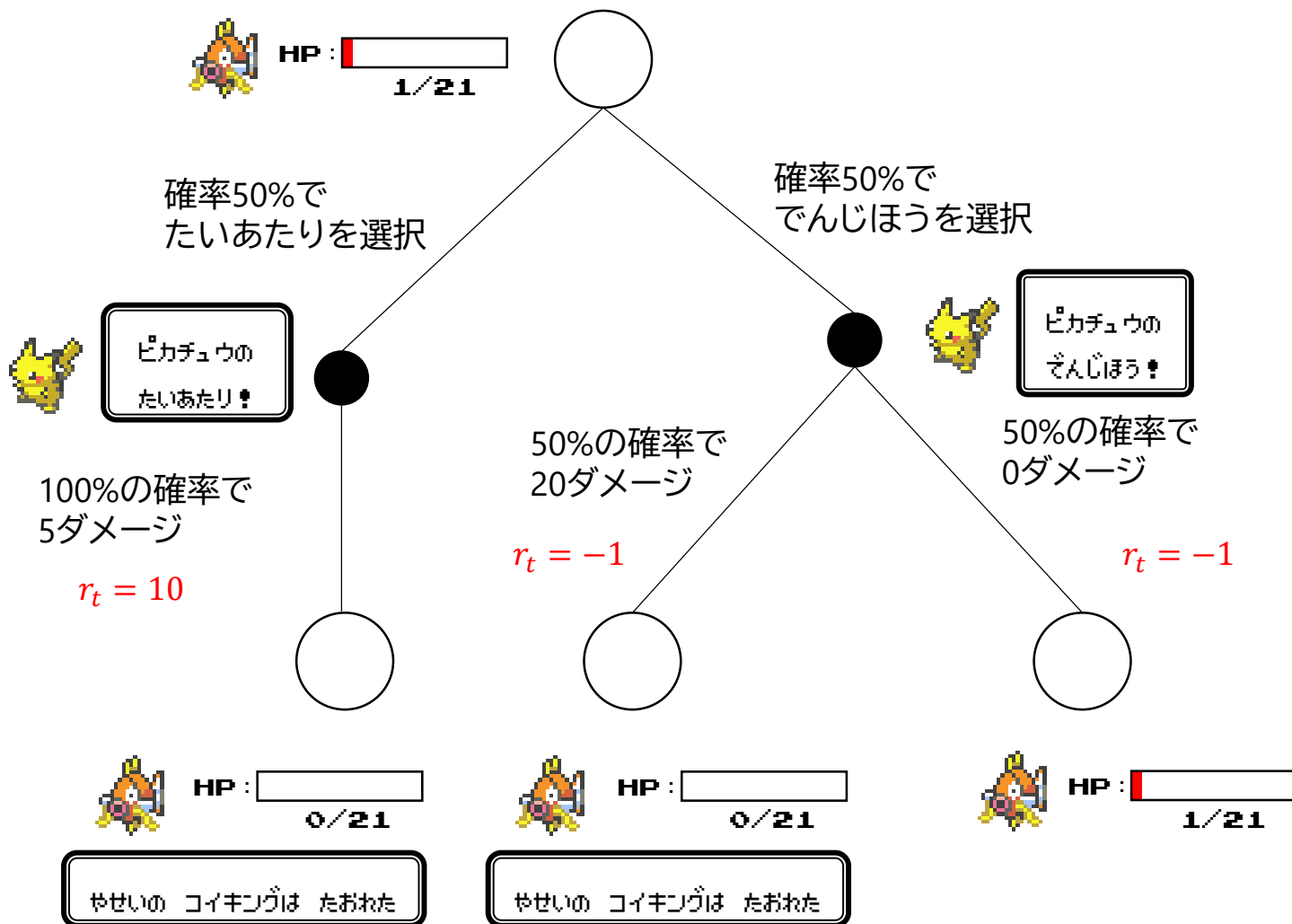
## 4. 価値関数と最適方策を推定する (動的計画法)

- 反復方策評価
- 方策反復(Policy Iteration)
- 価値反復(Value Iteration)

# ベルマン方程式を解いてみる③

45

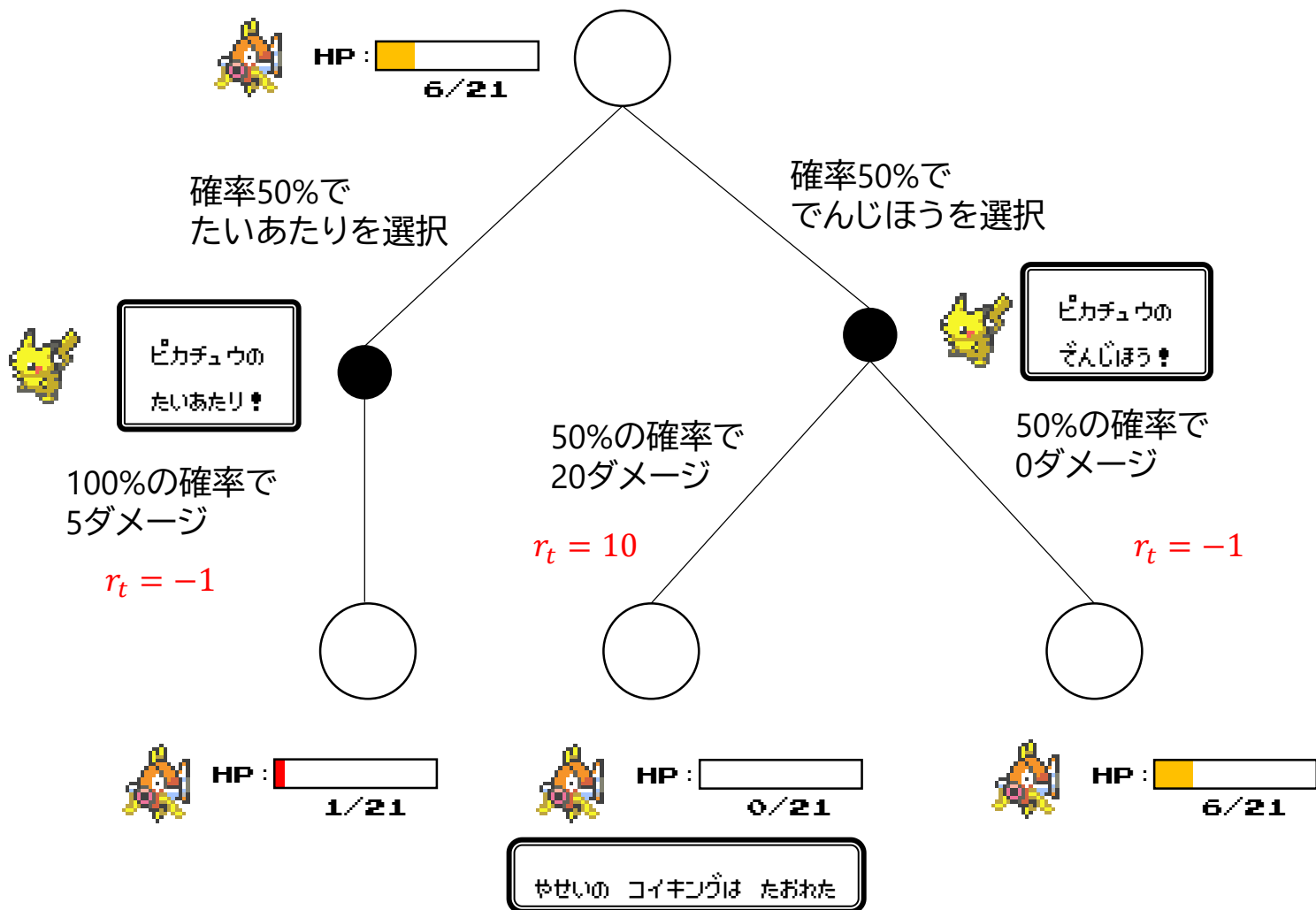
例:  $\pi(T) = \frac{1}{2}, \pi(D) = \frac{1}{2}$  の場合. すなわち, ランダムに技を選択する場合.  $\gamma = 1$  として割引なし.



# ベルマン方程式を解いてみる③

46

例:  $\pi(T) = \frac{1}{2}, \pi(D) = \frac{1}{2}$  の場合. すなわち, ランダムに技を選択する場合.  $\gamma = 1$  として割引なし.



残りHPが6のとき(左のバックアップ線図)

$$\begin{aligned} v_R(6) &= 0.5 \times 0.5 \times (-1 + v_R(1)) \\ &+ 0.5 \times 0.5 \times (10 + v_R(0)) \\ &+ 0.5 \times 0.5 \times (-1 + v_R(6)) \end{aligned}$$

整理して解くと

$$v_R(6) = \frac{7 + 2v_R(1)}{3} = \frac{79}{9} = 8.77 \dots$$

同様にして

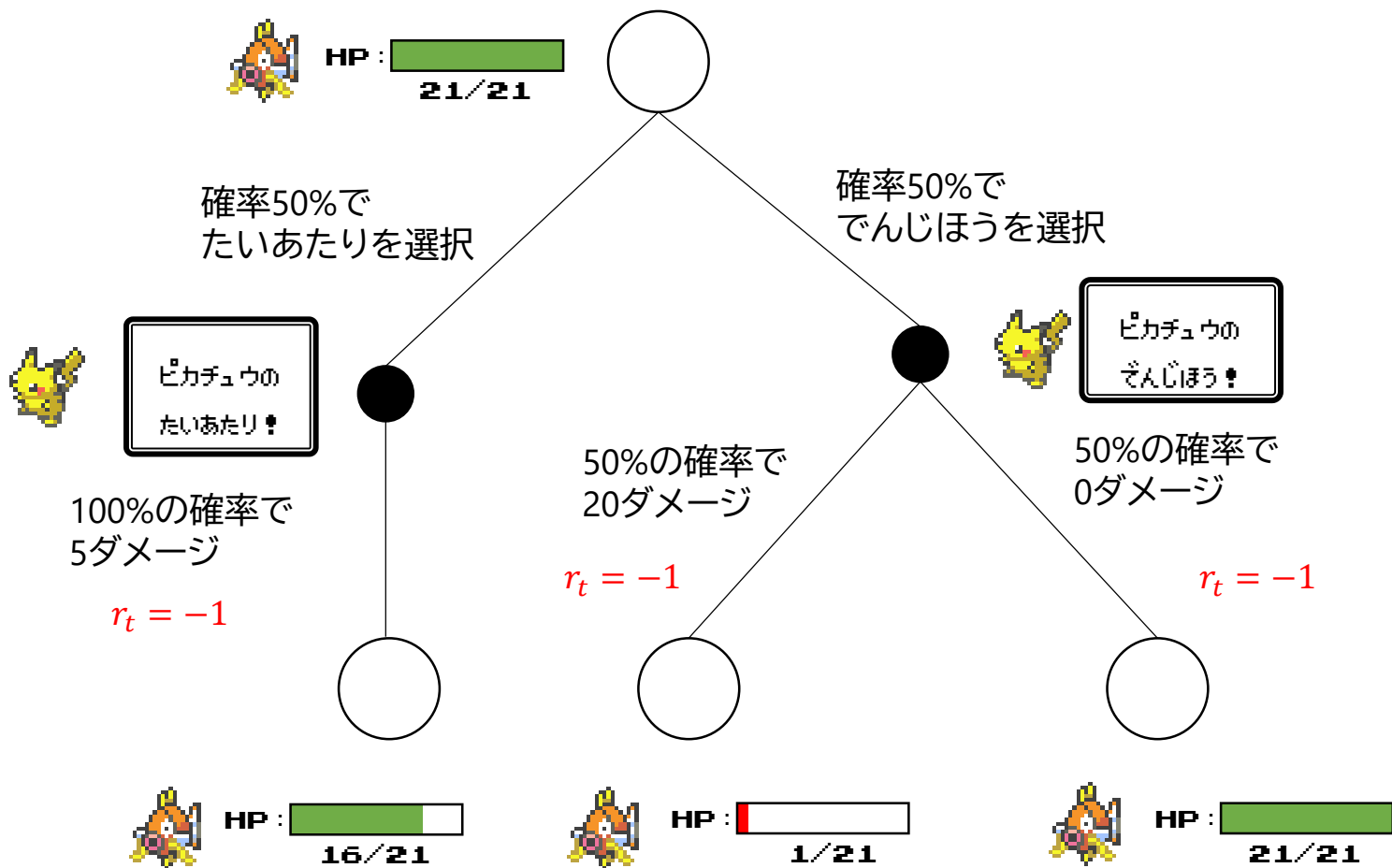
$$v_R(11) = \frac{7 + 2v_R(6)}{3} = \frac{221}{27} = 8.18 \dots$$

$$v_R(16) = \frac{7 + 2v_R(11)}{3} = \frac{631}{81} = 7.79 \dots$$

# ベルマン方程式を解いてみる③

47

例:  $\pi(T) = \frac{1}{2}, \pi(D) = \frac{1}{2}$  の場合. すなわち, ランダムに技を選択する場合.  $\gamma = 1$  として割引なし.



残りHPが21のとき(左のバックアップ線図)

$$\begin{aligned} v_R(21) &= 0.5 \times 0.5 \times (-1 + v_R(16)) \\ &+ 0.5 \times 0.5 \times (-1 + v_R(1)) \\ &+ 0.5 \times 0.5 \times (-1 + v_R(21)) \end{aligned}$$

整理して解くと

$$v_R(21) = \frac{7 + 2v_R(1) + v_R(21)}{3} = \frac{1721}{243} = 7.08 \dots$$

結果をまとめると  
右の表のとおり

$s$	$v_R(s)$
21	7.08
16	7.79
11	8.19
6	8.78
1	9.67
0	0

- これ以上ややこしい設定で、価値関数を解析的に求めることは面倒
- 強化学習アルゴリズムを用いて、繰り返し計算によって価値関数を推定する方法を考えよう
- まず、もっとも単純なアルゴリズムである動的計画法(DP, Dynamic Programming)を紹介する



- ベルマン方程式を、“推定した状態価値関数”の更新則として用いる。  
すなわち

$$\begin{aligned} v_{k+1}(s) &:= \mathbb{E}_{\pi}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_k(s')] \end{aligned}$$

次回( $k+1$ 回目)の,  
状態 $s$ における価値関数の推定値

現在( $k$ 回目)の,  
状態 $s'$ における価値関数の推定値

- これを、取りうるすべての状態  $s$  について計算し、繰り返し更新していくと、推定した状態価値関数は、やがて真の値に収束することが知られている

$$\begin{aligned} v_{\pi}(s) &= E_{\pi}[G_t | S_t = s] \\ &= E_{\pi} \left[ R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s \right] \\ &= \sum_a \pi(a|s) \left\{ \sum_{s'} p(s'|s, a) \{ r(s, a, s') + \gamma v_{\pi}(s') \} \right\} \end{aligned}$$

1. 取りうるすべての状態 $s$ に対して, すべての行動 $a$ を起こしてみて, 報酬を計算する
2. 得られた報酬を用いて, 価値関数の予測値 $v_k$ を更新していく
3. 以上を,  $v_k$ が収束するまで繰り返す

第2章でベルマン方程式を手計算で解いた方法と似ているが,  
それよりも少し愚直に手間を掛けて計算するアルゴリズム

すべての $s$ について $v(s) = 0$

収束するまで繰り返し:

各 $s \in S$ について繰り返し:

$$v(s) \leftarrow \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v(s)]$$

## 演習

MATLABで作成したサンプルプログラムpokemon\_DP1.mに, 上記アルゴリズムを実装している. 理論と実装の関係を確認した上で, 様々な問題設定について動的計画法を試してみよう. 例えば割引率を導入するとどうなるか? 方策を変えるとどうなるか?

※pokemon\_DP1.mは次ページにも示している

## MATLABで作成したサンプルプログラムpokemon\_DP1.m

```
1  clc
2  clear
3
4  % 環境を定義する
5  states = [0, 1, 6, 11, 16, 21]; % 状態(コイキングの残りHP)の集合
6  actions = ['T', 'D']; % 行動(ピカチュウの出す技)の集合
7  damages = [5, 20]; % 技ごとのダメージ量
8  accuracy = [1, 0.5]; % 技ごとの命中率
9  discount = 1; % 割引率
10
11 % 方策(技を選択する確率)を定義する
12 policy(1,:) = [0, 1]; % 残りHPが0のとき
13 policy(2,:) = [0, 1]; % 残りHPが1のとき
14 policy(3,:) = [0, 1]; % 残りHPが6のとき
15 policy(4,:) = [0, 1]; % 残りHPが11のとき
16 policy(5,:) = [0, 1]; % 残りHPが16のとき
17 policy(6,:) = [0, 1]; % 残りHPが21のとき
18
19 % 状態価値関数を定義する
20 value = zeros(1, length(states)); % 初期値は全て0
21
22 % 収束判定の閾値
23 delta = 1e-6;
```

上記設定での実行結果:

```
value =
      0    10.0000     9.0000     9.0000     9.0000     8.0000

i=24までで収束しました
fx >> |
```

繰り返し計算の結果, 解析解と同じ値に収束

```
25 % 繰り返し計算によって, 状態価値関数を推定する
26 for i = 1:10000 % 収束するのに十分な回数を繰り返す
27     value_old = value;
28     for i_s = 2:length(states) % v(s=0)=0は確定しているので, i_s=2からスタート
29         v = 0;
30         for i_a = 1:length(actions)
31             % 技が命中する場合
32             next_s = states(i_s) - damages(i_a);
33             if next_s < 0
34                 next_s = 0; % HPをマイナスにしない
35             end
36             v = v + policy(i_s, i_a) * accuracy(i_a) * (reward(next_s) + discount * value(states == next_s));
37
38             % 技を外した場合
39             next_s = states(i_s);
40             v = v + policy(i_s, i_a) * (1 - accuracy(i_a)) * (reward(next_s) + discount * value(states == next_s));
41         end
42         value(i_s) = v;
43     end
44     value
45     if max(abs(value - value_old)) < delta
46         disp(['i=', num2str(i), 'までで収束しました'])
47         break
48     end
49 end
```

```
51 % 報酬関数を定義する
52 function r = reward(next_s)
53     % コイキングを撃破していれば報酬を得る. そうでなければペナルティ
54     if next_s == 0
55         r = 10;
56     else
57         r = -1;
58     end
59 end
```

## プログラムの解説

```
25 % 繰り返し計算によって、状態価値関数を推定する
26 for i = 1:10000 % 収束するのに十分な回数を繰り返す
27     value_old = value;
28     for i_s = 2:length(states) % v(s=0)=0は確定しているので、i_s=2からスタート
29         v = 0;
30         for i_a = 1:length(actions)
31             % 技が命中する場合
32             next_s = states(i_s) - damages(i_a);
33             if next_s < 0
34                 next_s = 0; % HPをマイナスにしない
35             end
36             v = v + policy(i_s,i_a)*accuracy(i_a)*(reward(next_s) + discount*value(states == next_s));
37
38             % 技を外した場合
39             next_s = states(i_s);
40             v = v + policy(i_s,i_a)*(1-accuracy(i_a))*(reward(next_s) + discount*value(states == next_s));
41         end
42         value(i_s) = v;
43     end
44     value
45     if max(abs(value - value_old)) < delta
46         disp(['i=',num2str(i),'までで収束しました'])
47         break
48     end
49 end
```

それぞれの状態 $s$ について、  
すべての行動 $a$ を取った場合、  
可能な全ての次状態 $s'$ の価値 $v(s')$ を用いて  
ベルマン方程式で価値 $v(s)$ を更新する

今回の問題では、  
行動(ピカチュウの技)を選択したあと  
次の状態(コイキングの残りHP)は  
確率的に決定する。  
技が当たった場合と外れた場合  
それぞれの状態遷移について  
計算を行っている

予め環境の全てについて知っていないと、このアルゴリズムは適用できない  
(技のダメージや命中率などがわからない場合はプログラムが組めない)

- ある決定論的な方策 $\pi$ (それぞれの状態に対して100%の確率で特定の行動を取る)を与える
- 全ての状態において, 全ての取りうる行動への変更を考え, 各状態で行動価値関数 $q_\pi(s, a)$ が最良になる行動に更新する

$$\begin{aligned}\pi'(s) &= \arg \max_a q_\pi(s, a) \\ &= \arg \max_a \sum_{s', r} p(s', r | s, a) \{r + \gamma v_\pi(s')\}\end{aligned}$$

- 改善した方策 $\pi'$ について再び価値関数 $v_{\pi'}(s)$ を算出し, さらに改善した方策を探す
- 状態と行動が有限個ならば, 有限回の繰り返しで最適方策に収束する

1. 現在の方策 $\pi$ について, 状態価値関数 $v_\pi(s)$ を繰り返し計算によって推定
2. 各状態 $s$ について, 全ての行動 $a$ を行い, より良い行動を選択する新しい方策 $\pi'$ を作る
3. 上記を, 更新した方策 $\pi'$ と前の方策 $\pi$ に変化がなくなるまで繰り返す

第3章で最適方策を手計算で解いた方法と似ているが, それよりも  
少し愚直に手間を掛けて計算するアルゴリズム

方策が収束するまで繰り返し:

各 $s \in S$ について状態価値関数を推定

各 $s \in S$ について繰り返し:

$$\pi \leftarrow \arg \max_a q(s, a)$$

## 演習

MATLABで作成したサンプルプログラムpokemon\_DP2.mに, 上記アルゴリズムを実装している.  
理論と実装の関係を確認した上で, 様々な問題設定について動的計画法を試してみよう.  
例えば割引率を導入するとどうなるか? 初期方策を変えるとどうなるか?

※pokemon\_DP2.mは次ページにも示している

## MATLABで作成したサンプルプログラムpokemon\_DP2.mの一部

```
25 tic
26 for i_q = 1:10000
27     % まず、繰り返し計算によって、現在の方策に対して状態価値関数を推定する
28     for i_v = 1:10000 % 収束するのに十分な回数を繰り返す
29         value_old = value;
30         for i_s = 2:length(states) % v(s=0)=0は確定しているので、i_s=2からスタート
31             v = 0;
32             for i_a = 1:length(actions)
33                 % 技が命中する場合
34                 next_s = states(i_s) - damages(i_a);
35                 if next_s < 0
36                     next_s = 0; % HPをマイナスにしない
37                 end
38                 v = policy(i_s,i_a)*accuracy(i_a)*(reward(next_s) + discount*value(states == next_s));
39             end
40             % 技を外した場合
41             next_s = states(i_s);
42             v = v + policy(i_s,i_a)*(1-accuracy(i_a))*(reward(next_s) + discount*value(states == next_s));
43         end
44         value(i_s) = v;
45     end
46     if max(abs(value - value_old)) < delta
47         break
48     end
49     pause(0.01)
50 end
```

```
52 % 次に、状態価値関数を用いて、方策を改善する
53 policy_old = policy;
54 for i_s = 2:length(states)
55     % 状態価値関数を用いて、各行動の価値を計算する
56     q = zeros(1, length(actions));
57     for i_a = 1:length(actions)
58         % 技が命中する場合
59         next_s = states(i_s) - damages(i_a);
60         if next_s < 0
61             next_s = 0; % HPをマイナスにしない
62         end
63         q(i_a) = accuracy(i_a)*(reward(next_s) + discount*value(states == next_s));
64     end
65     % 技を外した場合
66     next_s = states(i_s);
67     q(i_a) = q(i_a) + (1-accuracy(i_a))*(reward(next_s) + discount*value(states == next_s));
68 end
69 % 各行動の価値を用いて、方策を改善する
70 [max_v, max_a] = max(q);
71 policy(i_s,:) = 0;
72 policy(i_s,max_a) = 1; % 最大の価値を持つ行動を選択する
73 end
74 if max(abs(policy - policy_old)) < delta
75     break
76 end
77 pause(0.01)
78 end
79 toc
80
```

### 上記設定での実行結果:

経過時間は 0.330274 秒です。

得られた最適方策 [Tを選択する確率,Dを選択する確率] は

```
s = 1    [1, 0]
s = 6    [0, 1]
s = 11   [0, 1]
s = 16   [0, 1]
s = 21   [1, 0]
```

繰り返し計算の結果, 解析解と同じ値に収束

ここでpauseを挟まないと  
計算が速すぎて正確に経過時間を得られない  
(実際の運用では不要)



- 反復方策では, 方策を更新するたびに, その方策に対する状態価値関数を繰り返し計算によって求めている. そのため, 計算負荷が大きい
- 方策評価(方策更新計算)を途中で打ち切っても収束が証明されている手法が存在:**価値反復**
- 方策評価において, **最大の価値を取る行動だけを採用し続ける**

$$v_{k+1}(s) = \max_a \sum_{s', r} p(s', r | s, a) \{r + \gamma v_k(s')\}$$

最大になる行動だけ採用

- これでも最適な方策における状態価値関数 $v^*(s)$ に収束することが示されている
- あとは $v^*(s)$ を用いて最大の行動価値関数を取るような(Greedyな)行動を選択する方策を組み立てるだけ



1. 各状態 $s$ について, 推定した状態価値関数 $V(s)$ を, 最も価値関数が大きくなる行動 $a$ を選択して更新
2. 上記を $V(s)$ が収束するまで繰り返し
3. 収束した $V(s)$ を用いて, 価値が最大になる行動を選択する方策を作る

価値関数が収束するまで繰り返し:

各 $s \in S$ について繰り返し:

$$V(s) \leftarrow \max_a \sum_{s',r} p(s',a) \{r + \gamma V(s')\}$$

$\pi(s) = \arg \max_a \sum_{s',r} p(s',a) \{r + \gamma V(s')\}$ となる決定論的方策 $\pi^*$ を出力

## 演習

MATLABで作成したサンプルプログラムpokemon\_DP3.mに, 上記アルゴリズムを実装している. 理論と実装の関係を確認した上で, 様々な問題設定について動的計画法を試してみよう. 例えば割引率を導入するとどうなるか? 初期方策を変えるとどうなるか?

## MATLABで作成したサンプルプログラムpokemon\_DP3.mの一部

```
25 tic
26 % まず、繰り返し計算によって、現在の方策に対して状態価値関数を推定する
27 for i_v = 1:10000 % 収束するのに十分な回数を繰り返す
28     value_old = value;
29     for i_s = 2:length(states) % v(s=0)=0は確定しているので、i_s=2からスタート
30         v = zeros(1, length(actions));
31         for i_a = 1:length(actions)
32             % 技が命中する場合
33             next_s = states(i_s) - damages(i_a);
34             if next_s < 0
35                 next_s = 0; % HPをマイナスにしない
36             end
37             v(i_a) = policy(i_s,i_a)*accuracy(i_a)*(reward(next_s) + discount*value(states == next_s));
38
39             % 技を外した場合
40             next_s = states(i_s);
41             v(i_a) = v(i_a) + policy(i_s,i_a)*(1-accuracy(i_a))*(reward(next_s) + discount*value(states == next_s));
42         end
43         value(i_s) = max(v);
44     end
45     if max(abs(value - value_old)) < delta
46         break
47     end
48     pause(0.01)
49 end
```

```
51 % 次に、状態価値関数を用いて、方策を改善する
52 policy_old = policy;
53 for i_s = 2:length(states)
54     % 状態価値関数を用いて、各行動の価値を計算する
55     q = zeros(1, length(actions));
56     for i_a = 1:length(actions)
57         % 技が命中する場合
58         next_s = states(i_s) - damages(i_a);
59         if next_s < 0
60             next_s = 0; % HPをマイナスにしない
61         end
62         q(i_a) = accuracy(i_a)*(reward(next_s) + discount*value(states == next_s));
63
64         % 技を外した場合
65         next_s = states(i_s);
66         q(i_a) = q(i_a) + (1-accuracy(i_a))*(reward(next_s) + discount*value(states == next_s));
67     end
68
69     % 各行動の価値を用いて、方策を改善する
70     [max_v, max_a] = max(q);
71     policy(i_s,:) = 0;
72     policy(i_s,max_a) = 1; % 最大の価値を持つ行動を選択する
73
74     pause(0.01)
75 end
76 toc
```

### 上記設定での実行結果:

経過時間は 0.104205 秒です。  
得られた最適方策 [Tを選択する確率, Dを選択する確率] は

s = 1	[1, 0]
s = 6	[0, 1]
s = 11	[0, 1]
s = 16	[0, 1]
s = 21	[0, 1]

繰り返し計算の結果、解析解と同じ値に収束

価値関数を求めるループと最適方策を求めるループが入れ子構造になっていないことが価値反復のポイント！  
(方策反復と比較してみよう)

pokemon\_DP2.mの結果と比べて、  
計算にかかった時間は明らかに削減されている

- 強化学習の基本的なアルゴリズムとして、動的計画法を紹介した
- 状態価値関数は繰り返し計算によって推定できる
- 推定した状態価値関数を用いて、最適方策を推定できる(方策反復)
- 計算量を省略できる価値反復アルゴリズムも紹介した
- 今回紹介した動的計画法のアルゴリズムは、  
環境モデルが予め全て分かっている必要があるという弱点がある



## 5. 未知環境に対する学習 (モンテカルロ法)

- 動的計画法では、環境を全て知っている必要があった
- 例えば、最も価値の高い行動を選択し続けるアルゴリズム「価値反復」では、ある行動を起こしたとき、**どれだけの確率で、どの次状態に遷移するのか、予め知っていないと計算ができない**

$$v_{k+1}(s) = \max_a \sum_{s', r} \underbrace{p(s', r | s, a)}_{\text{これを予め知っている必要}} \{r + \gamma v_k(s')\}$$

これを予め知っている必要

- 環境が未知の場合は、最適行動を探索するどころか、価値関数の推定すらできない
- 例えば、ピカチュウが命中率と威力が不明な新たな技「ピカボルト」を覚えたとき、どうすれば最適方策を得られるだろう？

- 再び, ある方策に対して状態価値関数を推定する方法を考えるところから始める
- **モンテカルロ予測**では, 状態価値関数を今後得られる**累計報酬の平均値**として推定
  - 多数回の実験を行い平均値を取ることで, 未知の環境に対しても適用できる
  - 状態価値関数の本来の定義は, 「今後得られる**累計報酬の期待値**」
  - 理論上, 無限回の実験を行えば, 期待値 = 平均値 (大数の法則)



1. 1つのエピソード(初期状態から終端状態にいたるまで)を実験やシミュレーションによって生成
2. 得られたエピソードから, 各ステップに対して累計報酬を計算
3. 累計報酬の平均値を, 価値関数の推定値とする
4. 1~3を, できるだけ多くの回数繰り返す

できるだけ多く繰り返し:

エピソードの生成

$G \leftarrow 0$

エピソードの各ステップについて繰り返し

$G \leftarrow \gamma G + r$

$s$ が, エピソード内で初出現なら

$G$ を $Returns(s)$ に付け足し(足し算するのではない)

$V(s) \leftarrow average(Returns(s))$

今回の設定のポケモン問題であれば,  
コイキングのHPは減る一方だから  
この条件は必ず満たされる

これまでの実験結果をひたすら貯める箱

MATLABで作成したサンプルプログラムpokemon\_MC1.mに, 上記アルゴリズムを実装している.  
pokemon\_MC2.mでは, メモリ使用量を削減した平均値の算出を行っている.

## サンプルプログラムpokemon\_MC1.mの一部

```
% 方策を定義する.  
policy = 3;  
  
% 状態価値関数を定義する  
value = zeros(length(states),1); % 初期値は全て0  
  
% 平均値を求めるためにデータを貯めておく箱  
for i_s = 1:length(states)  
    R(i_s).G = [];  
end  
  
tic  
% 繰り返し計算によって、現在の方策に対して状態価値関数を推定する  
for i_v = 1:1e5 % たくさん繰り返す  
  
    s0 = round(rand()*max(states)); % 初期状態の残りHPをランダムに決定する  
    [sset,rset] = episode(s0,policy); % 1回の戦闘が終了するまで実行する  
  
    G = 0;  
    for i_s = length(sset):-1:1  
        s = sset(i_s);  
        G = G + discount * rset(i_s);  
        R(states==s).G = [R(states==s).G;G];  
        value(states==s) = mean([R(states==s).G]);  
    end  
  
    if discount == 1 && policy < 3 % この場合は解析解を求めている  
        error(i_v) = norm(value - value_true(:, policy));  
    end  
  
end  
toc
```

何らかの方法でエピソードを生成する  
アルゴリズム内に環境の情報が入っていないので、  
未知環境においてもこのアルゴリズムは適用可能

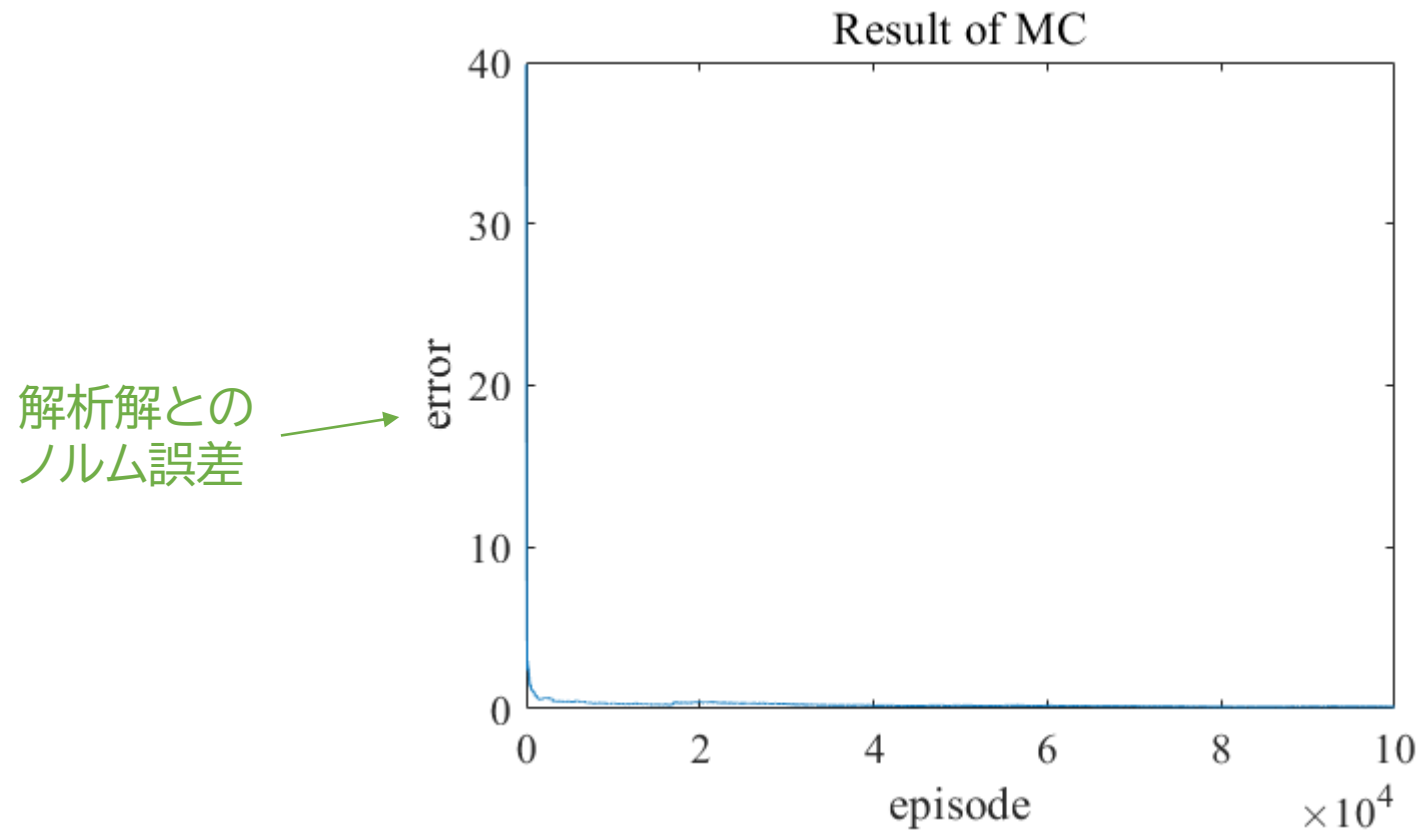
わざのダメージ設定によっては、何度繰り返しても  
全ての状態 $s$ が得られないことがある  
そのため、コイキングの残りHPを  
最大値以外でも試すようにしている

例によって終端状態から逆向きに計算している

蓄えたデータの平均値から状態価値関数を推定

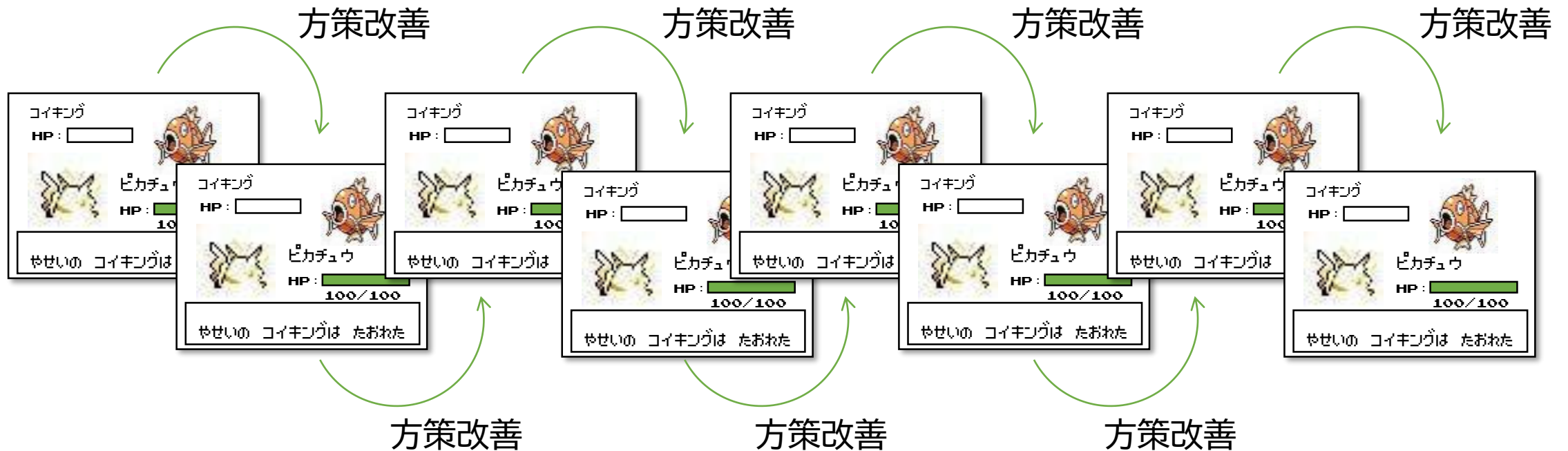


サンプルプログラムpokemon\_MC1.mを実行し、  
でんじほうのみを選択する方策に対する状態価値関数を推定した結果



- 理論値に近い価値関数の推定値が得られる
- より高い精度を得るためには、繰り返し回数を増やしていけばよいが、計算時間はその分増える

- 多くのエピソードを使って, 行動価値関数を推定できる
- 推定値をもとにして行動価値関数を作り, 最適な行動を求めていく
- 理論上は, 方策改善のたびに多数のエピソードを用いて方策を評価するのがよいが, それでは時間がかかりすぎてしまう
- 実際の運用では, エピソードごとに評価と改善を繰り返す



1. ランダムに状態 $s_0$ と行動 $a_0$ を選択(開始点探索, Exploring Starts)
2. 現在の方策 $\pi$ にそって,  $s_0$ と $a_0$ から始まるエピソードを生成
3. エピソード内の各 $s$ について, 平均値として行動価値関数 $Q(s, a)$ を推定
4. 行動価値関数を用いて, Greedyな方策 $\pi$ に更新
5. 1~4をできるだけ多く繰り返す

できるだけ多く繰り返し:

ランダムに状態 $s_0$ と行動 $a_0$ を選択

方策 $\pi$ に基づいて,  $s_0$ と $a_0$ から始まるエピソードを生成

$G \leftarrow 0, n \leftarrow 0$

エピソードの各ステップについて繰り返し

$G \leftarrow \gamma G + r$

$(s, a)$ が, エピソード内で初出現なら

$Q(s, a) \leftarrow (Q(s, a)n_{s,a} + G)/(n_{s,a} + 1)$  ← メモリを節約しながら  
平均値を計算

$\pi(s) \leftarrow \arg \max_a Q(s, a)$

$n_{s,a} \leftarrow n_{s,a} + 1$

## サンプルプログラムpokemon\_MC3.mの一部

```
20 % 繰り返し計算によって、現在の方策に対して状態価値関数を推定する
21 for i_v = 1:1e6 % たくさん繰り返す
22
23     s0 = round(rand()*(max(states)-1))+1; % 初期状態の残りHPをランダムに決定する
24     a0 = round(rand()*(max(actions)-1))+1; % 初期行動をランダムに決定する
25     [sset,aset,rset] = episode(s0,a0, policy,states); % 1回の戦闘が終了するまで実行する
26
27     G = 0;
28     for i_s = length(sset):-1:1
29         s = sset(i_s);
30         a = aset(i_s);
31         G = G + discount * rset(i_s);
32         cnt(states==s,actions==a) = cnt(states==s,actions==a) + 1;
33         q(states==s,actions==a) = (q(states==s,actions==a)*(cnt(states==s,actions==a)-1) + G)/cnt(states==s,actions==a); % 平均値を求める
34         [max_q, max_a] = max(q(states==s,:));
35         policy(states==s,:) = zeros(1,length(actions));
36         policy(states==s,max_a) = 1;
37     end
38 end
```

適当な初期値からスタートして、すべての場合を試す  
また、1回目の行動だけは現在の方策に関係なく  
ランダムに試してみて、新たな可能性を探る

行動価値関数は平均値として求める

現在推定している行動価値関数が最大になるように、  
方策を更新する

上記プログラムを実行すると、理論値に近い価値関数の推定値と、それに基づいた最適方策を得る  
より高い精度を得るためには、繰り返し回数を増やしていけばよいが、計算時間はその分増える



## 6. Q学習とSARSA (TD法)

## 【DP法】

- ベルマン方程式を活用して、各状態における状態価値関数を推定
- 環境が全て分かっていないと実行できない

## 【モンテカルロ法】

- 未知環境でも、たくさんのサンプルデータを用いて、状態価値関数を推定
- ベルマン方程式に示される、状態と状態の関係を全く活用していない

Q 両方の良いところを取ってきたような強化学習は可能だろうか？

A 時間的差分(**Temporal Differential, TD**)学習なら可能！

- 環境モデルを必要とせず、経験から学ぶ
- ベルマン方程式を活用して、推定値を他の推定に用いる(ブートストラップ)ことで、エピソードが終了する前に計算可能

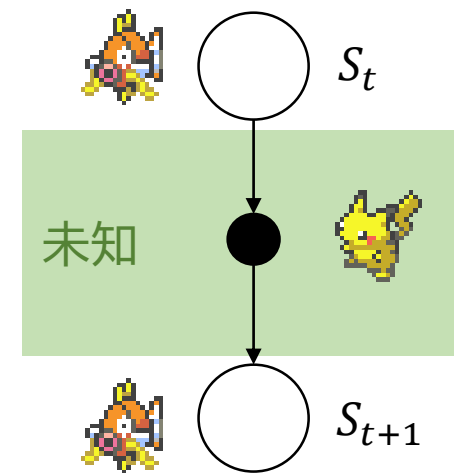
- ここでも, ある方策に対して状態価値関数を推定することから始める
- ベルマン方程式  $v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$  を参考に, 実験(シミュレーション)中に報酬を受け取り, 次状態が確定した段階で,

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

によって状態価値関数の平均値を取ることなく,  
推定値を逐次的に更新していく

## TD誤差

現在の推定値 $V(S_t)$ と, より良い推定値 $R_{t+1} + \gamma V(S_{t+1})$ との誤差



## 特徴

- モンテカルロ法と同様に, 環境モデルが不明でも使える
- モンテカルロ法と違って, エピソードが終了するのを待つ必要がない

pokemon\_TD1.mが, TD(0)法とも呼ばれるTD予測のサンプルプログラムである

## サンプルプログラムpokemon\_TD1.mの一部

```
% ステップサイズパラメータ
```

```
alpha = 0.1;
```

```
% 状態価値関数
```

```
v = zeros(length(states),1); % 初期値は全て0
```

```
tic
```

```
% 繰り返し計算によって、現在の方策に対して状態価値関数を推定する
```

```
for i_v = 1:1e5 % たくさん繰り返す
```

```
    s = round(rand()*(max(states)-1))+1; % 初期状態の残りHPをランダムに決定する
```

```
    while s > 0
```

```
        a = find(policy(states==s,:),1);
```

```
        next_s = battle(s,a);
```

```
        r = reward(next_s);
```

```
        v(states==s) = v(states==s) + alpha*(r + discount*v(states==next_s) - v(states==s));
```

```
        s = next_s;
```

```
    end
```

```
    if discount == 1 && i_action < 3 % この場合は解析解を求めている
```

```
        error(i_v) = norm(v - v_true(:, i_action));
```

```
    end
```

```
end
```

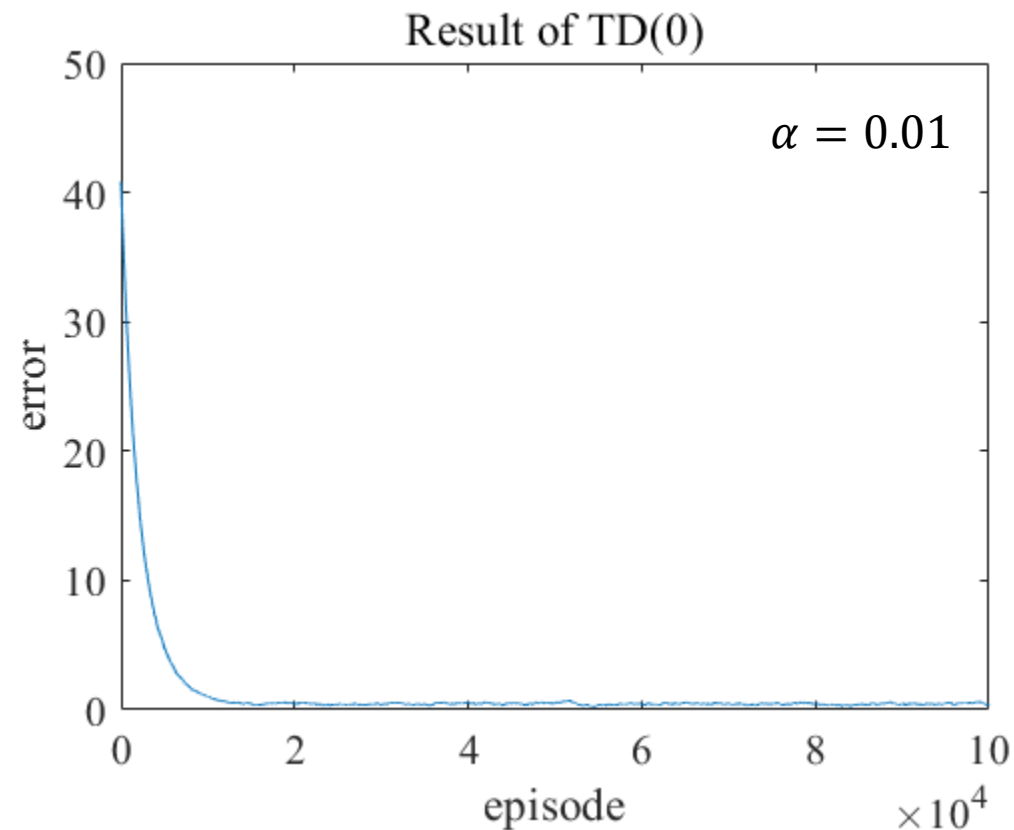
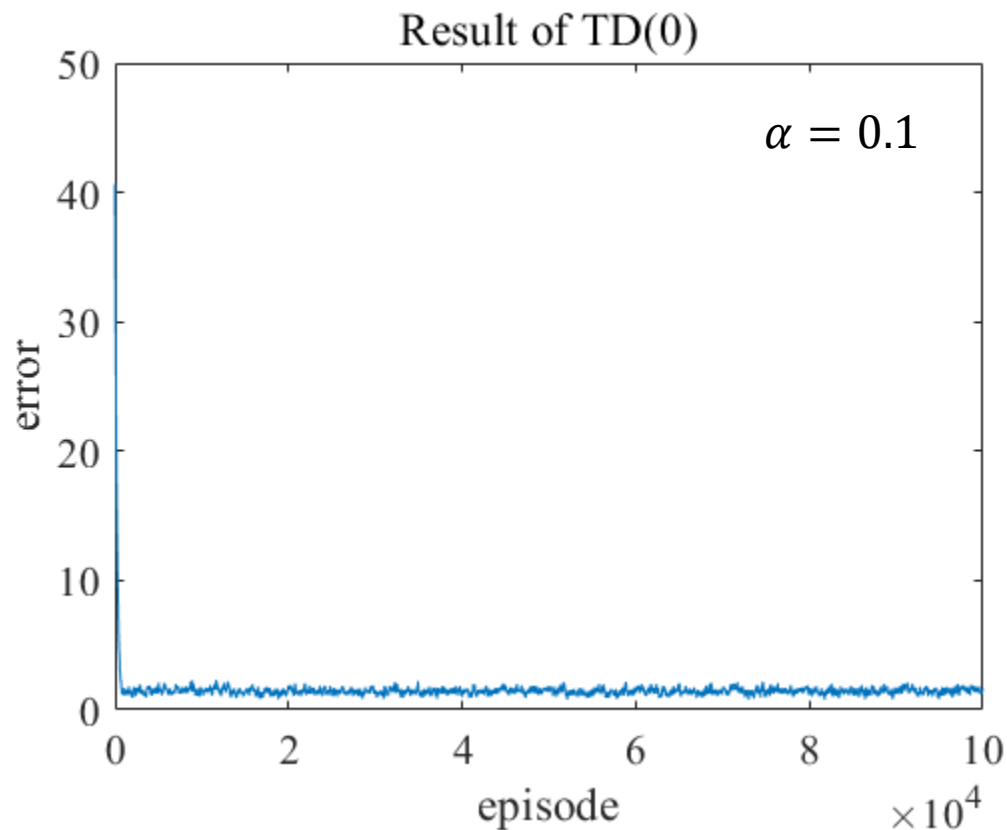
```
toc
```

エピソードが終了するのを待つことなく、  
1ターンの行動が終了した時点で  
状態価値関数を更新していく



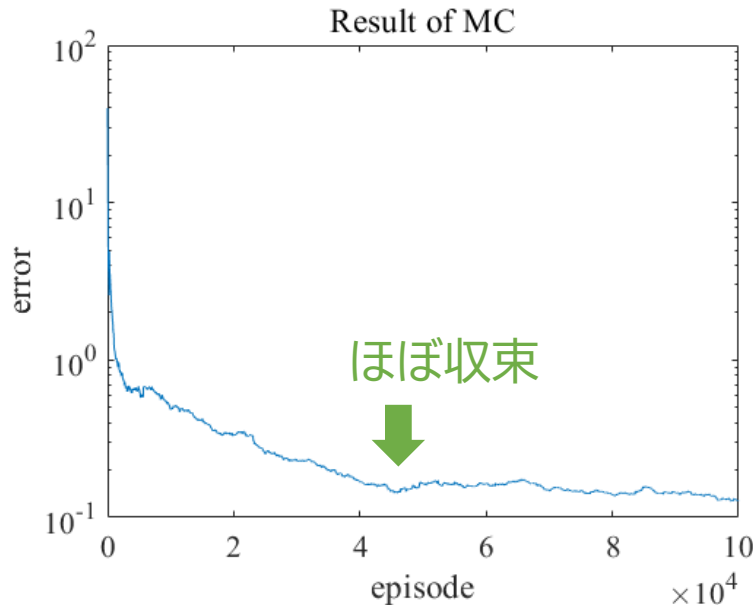


サンプルプログラムpokemon\_TD1.mを実行し、  
でんじほうのみを選択する方策に対する状態価値関数を推定した結果

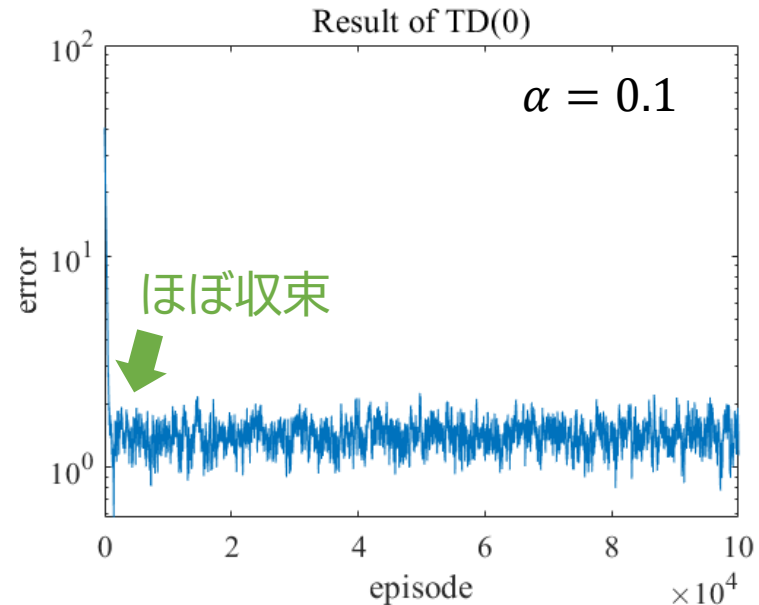


- 理論値に近い価値関数の推定値が得られる
- 誤差はある程度まで下がるが、ステップサイズパラメータ $\alpha$ が一定であれば、最新の結果に応じていつまでも上下し続ける
- $\alpha$ が小さければ、収束は遅くなるが、誤差はより小さな値に収束する
- 計算が進むほど $\alpha$ を小さくするように調整すれば、誤差はやがて0に収束する

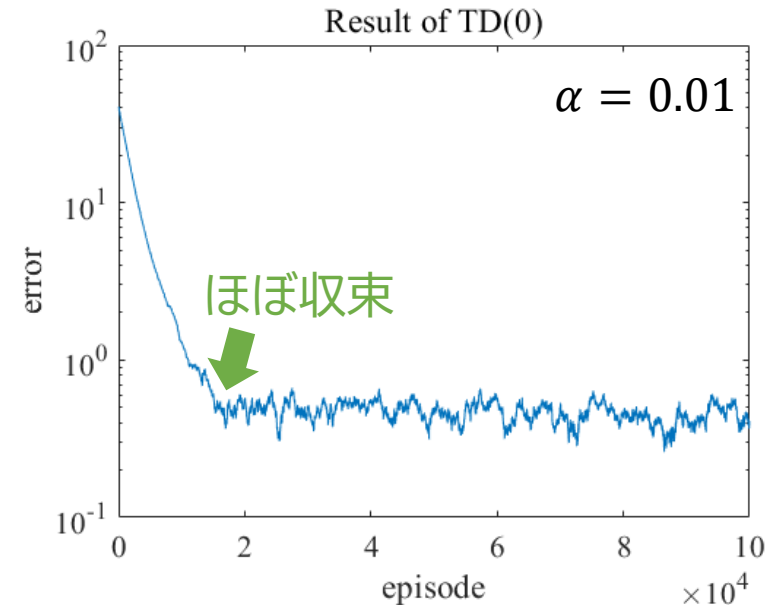
- TD法はエピソードの途中に状態価値関数を更新し続けるので、長い時間のかかるエピソードに対しても適用できる
- 収束速度の違いについて数学的な証明はないが、一般的にはTD法のほうが速い
- ただし、 $\alpha$ 一定の条件下では、TD法では誤差0に収束できない



モンテカルロ法の計算結果



TD法の計算結果

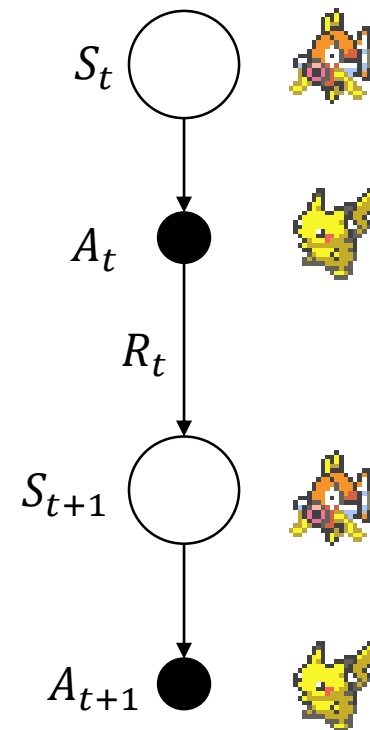


- TD予測にもとづいて, 最適方策を得る方法を考える
- 基本的には方策 $\pi$ に従い(On-policy), たまに異なる行動を試してみる
- 行動価値関数に次の更新則を適用する:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

TD誤差から行動価値関数の更新量を決定

- エピソードの完了まで待たず,  $Q(S_t, A_t)$ が更新されるたびに最大の $Q(S_t, A_t)$ を取るようの方策を改善していく
- このアルゴリズムは5つの情報( $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ )を用いるので**SARSA法**と呼ぶ



1. 現在の行動価値関数が最大になるように(小さい確率でランダムに,)  $s$  で取る行動  $A$  を選択
2. 次の状態  $s'$  と, 得られる報酬  $R$  を観測
3. 更新則  $Q(s_t, A_t) \leftarrow Q(s_t, A_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1}, A_{t+1}) - Q(s_t, A_t)]$
4. 1~3をエピソードが終わるまで繰り返し
5. 1~4を何エピソードか繰り返す

初期状態を設定

$Q$ に基づいて, (小さい確率でランダムに,)  $s$  で取る行動  $A$  を選択

エピソード終了まで各ステップについて繰り返し

行動  $A$  を取り, 報酬  $R$  と次状態  $s'$  を観測

$Q$  を最大にするように(小さい確率でランダムに), 次の行動  $A$  を選択

$Q(s, A) \leftarrow Q(s, A) + \alpha[R + \gamma Q(s', A') - Q(s, A)]$

$s \leftarrow s', A \leftarrow A'$

SARSA法をpokemon\_TD2.mに実装している. どのような実装になるか確認しよう.

## サンプルプログラムpokemon\_TD2.mの一部

% ステップサイズパラメータ

alpha = 0.2;

% イブシロン

e = 0.2;

% 状態価値関数

q = zeros(length(states),length(actions)); % 初期値は全て0

for i\_q = 1:num % たくさん繰り返す

    s = max(states); % 初期状態の残りHP

    if rand() < e

        % 小さい確率eでランダムに行動を選択する

        a = randi(length(actions));

    else

        % 最大の価値を持つ行動を選択する

        [~,a] = max(q(states==s,:));

    end

    while s > 0

        next\_s = battle(s,a);

        if rand() < e

            % 小さい確率eでランダムに行動を選択する

            next\_a = randi(length(actions));

        else

            % 最大の価値を持つ行動を選択する

            [~,next\_a] = max(q(states==s,:));

        end

        r = reward(next\_s);

        q(states==s,actions==a) = q(states==s,actions==a) + alpha\*(r + discount\*q(states==next\_s,actions==next\_a) - q(states==s,actions==a));

        total\_rewards(i\_q) = total\_rewards(i\_q) + discount \* r;

        s = next\_s;

        a = next\_a;

        steps(i\_q) = steps(i\_q) + 1;

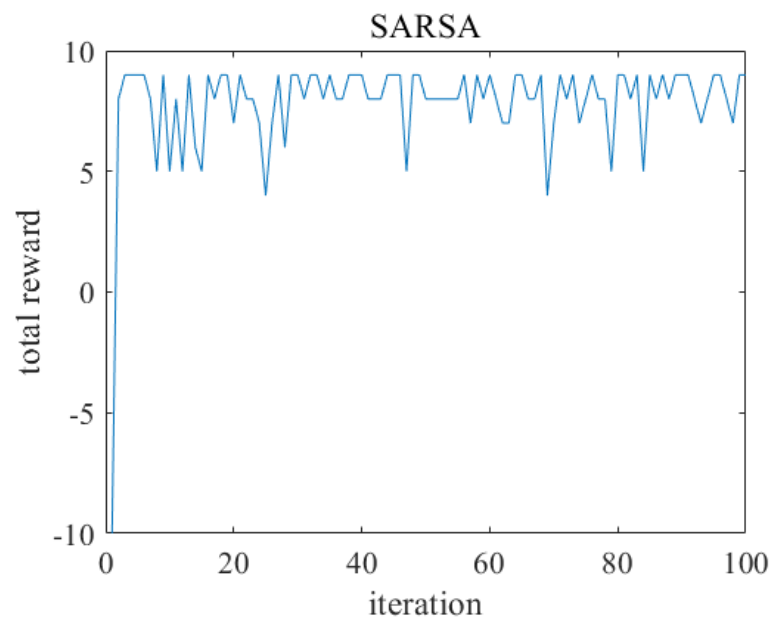
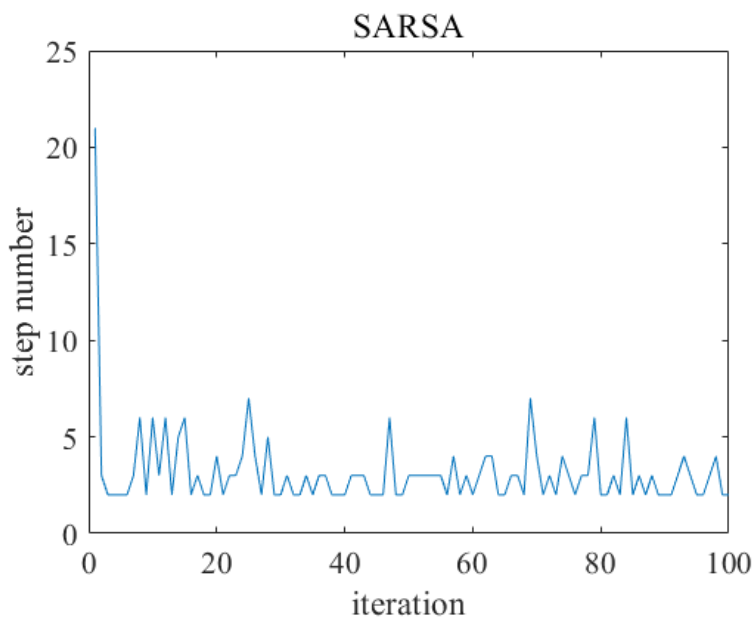
    end

end

行動価値関数にもとづき、  
現時点で最適な行動を取る  
小さい確率で  
ランダムな行動を取る  
( $\epsilon$ -Greedy)

エピソードが終了するのを待つことなく、  
1ターンの行動が終了した時点で  
現在と次状態の情報をもとに  
行動価値関数を更新していく

- たいあたりのダメージを1に変更して計算した結果
- $\alpha = 0.2, \varepsilon = 0.2$ と設定している
- 100回しかエピソードを回していないものの、方策が改善された結果、すでに良い報酬が得られるようになっている
- $\alpha, \varepsilon$ を一定値としているので、最終的に方策が収束することはない

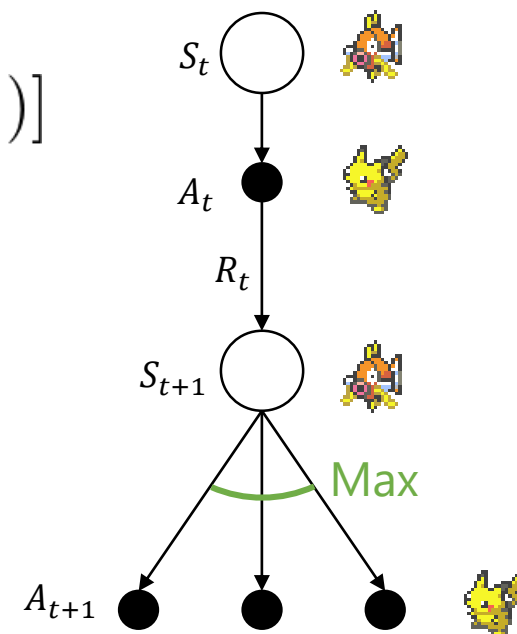


- SARSAより優れた方法として, **Q学習**が提案されている
- 現在の行動 $A_t$ は方策に従うか, 小さい確率で新しい行動を試す( $\epsilon$ -Greedy)
- 行動 $A_{t+1}$ は**方策に従わず(Off-policy)**, **最も行動価値関数が大きくなるものを選択する**ものとする。これでSARSAよりも効率よく真の値に近づいていけるはず
- 行動価値関数の更新則は

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \underbrace{\gamma \max_a Q(S_{t+1}, a)}_{\text{TD誤差を考えると、今後の価値が最大になるものを選ぶ}} - Q(S_t, A_t)]$$

TD誤差を考えると、今後の価値が最大になるものを選ぶ

- エピソードの完了まで待たず,  $Q(S_t, A_t)$ が更新されるたびに最大の $Q(S_{t+1}, a)$ を取るよう方策を改善していく



1. 現在の行動価値関数が最大になるように(小さい確率でランダムに),  $s$ で取る行動 $A$ を選択
2. 次の状態 $s'$ と, 得られる報酬 $R$ を観測
3. 更新則  $Q(s_t, A_t) \leftarrow Q(s_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, A_t)]$
4. 1~3をエピソードが終わるまで繰り返す
5. 1~4を何エピソードか繰り返す

初期状態を設定

エピソード終了まで各ステップについて繰り返す

$Q$ を最大にするように(小さい確率でランダムに), 状態 $s$ における行動 $A$ を選択

行動 $A$ を取り, 報酬 $R$ と次状態 $s'$ を観測

$$Q(s, A) \leftarrow Q(s, A) + \alpha [R + \gamma \max_a Q(s', a) - Q(s, A)]$$

$$s \leftarrow s'$$

Q学習をpokemon\_TD3.mlに実装している. どのような実装になるか確認しよう.



## サンプルプログラムpokemon\_TD3.mの一部

```
% ステップサイズパラメータ
```

```
alpha = 0.2;
```

```
% イプシロン
```

```
e = 0.2;
```

```
% 状態価値関数
```

```
q = zeros(length(states),length(actions)); % 初期値は全て0
```

```
for i_q = 1:num % たくさん繰り返す
```

```
    s = max(states); % 初期状態の残りHP
```

```
    while s > 0
```

```
        if rand() < e
```

```
            % 小さい確率eでランダムに行動を選択する
```

```
            a = randi(length(actions));
```

```
        else
```

```
            % 最大の価値を持つ行動を選択する
```

```
            [~,a] = max(q(states==s,:));
```

```
        end
```

```
        next_s = battle(s,a);
```

```
        r = reward(next_s);
```

```
        q(states==s,actions==a) = q(states==s,actions==a) + alpha*(r + discount*max(q(states==next_s,:)) - q(states==s,actions==a));
```

```
        total_rewards(i_q) = total_rewards(i_q) + discount * r;
```

```
        s = next_s;
```

```
        steps(i_q) = steps(i_q) + 1;
```

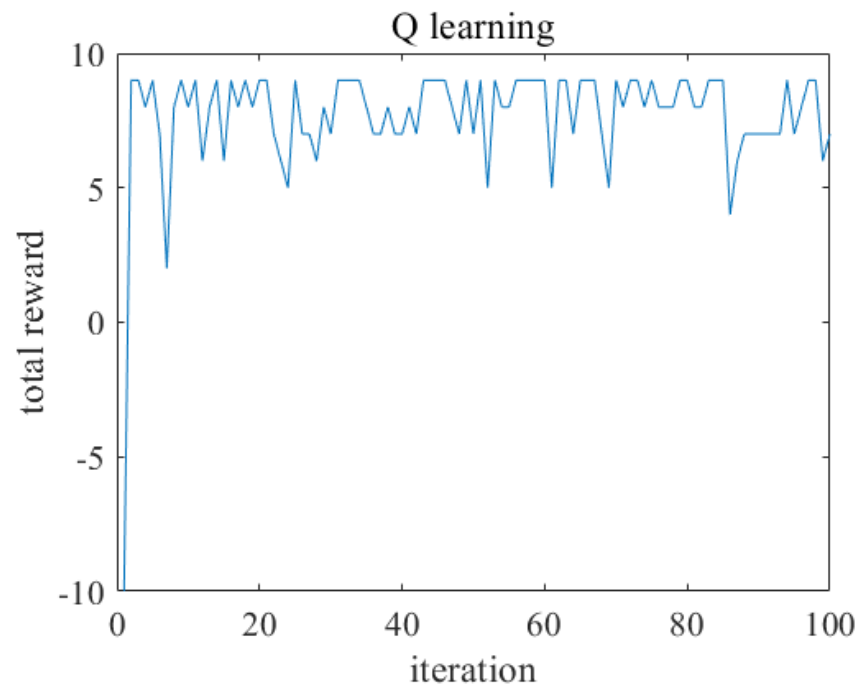
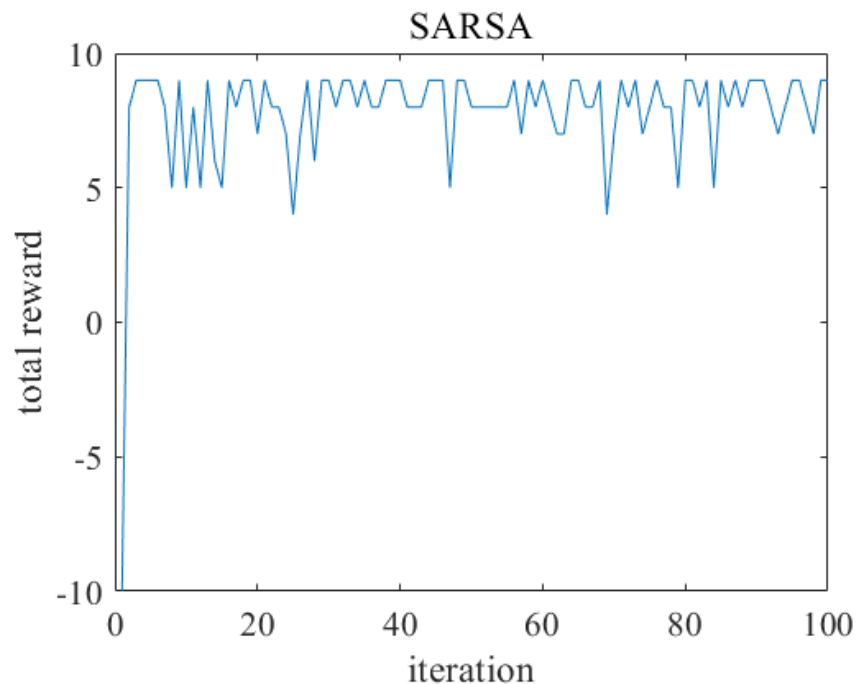
```
    end
```

```
end
```

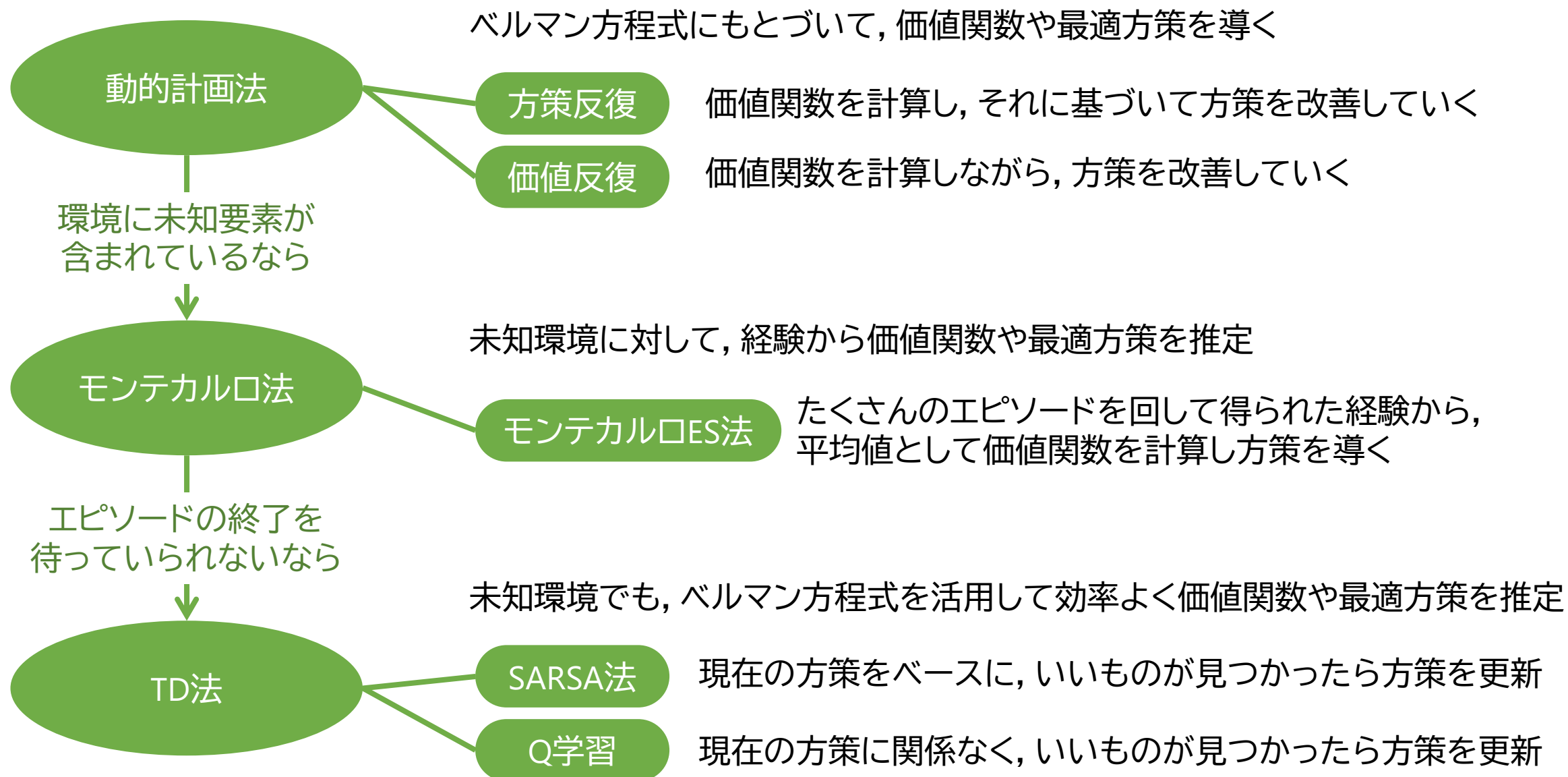
行動価値関数にもとづき、  
現時点で最適な行動を取る  
小さい確率で  
ランダムな行動を取る  
( $\epsilon$ -Greedy)

エピソードが終了するのを待つことなく、  
1ターンの行動が終了した時点で  
行動価値関数を更新していく

- いずれもたいあたりのダメージを1に変更して,  $\alpha = 0.2, \varepsilon = 0.2$ と設定した結果
- 今回の問題設定では, いずれも遜色ない結果が得られている
  - 今回の実装ではSARSAもランダム要素を含み, 完全にオンポリシーになっていないことが影響
  - Q学習のほうが結果が悪くなることもある



- 環境モデルが未知でも使える強化学習のアルゴリズムとして、**モンテカルロ法**と**TD法**を紹介した
  - いずれも、環境モデルが完全に与えられていない状況で、価値関数を推定し最適方策を導ける
- モンテカルロ法では、多くのエピソードを収集し、**平均値を取ることで価値関数を推定**するが、エピソードが完了するまで推定値の更新は行われない
- TD法では、エピソードの完了を待つことなく、各ステップにおいて**TD誤差を求め、価値関数を更新**していく
  - **SARSA法**では、現在の推定方策にもとづいて(**オンポリシー**)、ステップごとに行動価値関数を改善
  - **Q学習**では、現在の推定方策に依存することなく(**オフポリシー**)、効率よく行動価値関数を改善



- [1] R. Sutton & Barto “Reinforcement Learning : An Introduction Second Edition,” 2018 (邦訳「強化学習 第2版」, 森北出版, 2022)
- [2] 強化学習の基礎と深層強化学習(東京大学 松尾研究室 深層強化学習サマースクール講義資料)  
<https://www.slideshare.net/ShotaImai3/rlssdeepreinforcementlearning>
- [3] 中井:「ITエンジニアのための強化学習理論入門」, 技術評論社, 2020