



# ポケモンで きょうか がくしゅう 2 きそへん

ポケモンを例題として, 動的計画法を理解する  
名古屋工業大学 助教 上村知也



## 1. 問題設定(前回の復習)

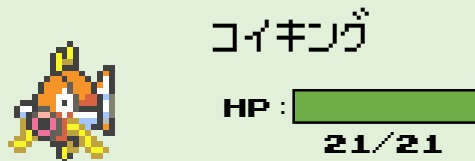


## 2. 価値関数と最適方策を推定する(動的計画法)

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

# 1. 問題設定(前回の復習)

- 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
  - 乱数によるダメージの増減は考えない
  - 状態異常や能力値変化は考慮しない, あるいはその影響は無視できる
  - 天候は考慮しない
  - PPは尽きないものとする
  - 道具は使用できない
  - ポケモンは交代できない
  - せいかくや持ち物その他による能力値や状態の変化は発生しない



わざ：

はねる      めいちゅう ○    げんじほう ○

HP:21

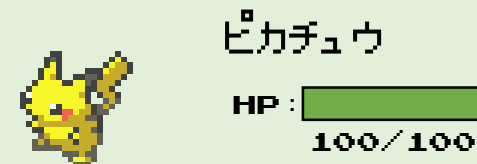
持ち物なし

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

敵のポケモン



わざ：

たいあたり    めいちゅう    100    げんじほう    5

でんじほう    めいちゅう    50    げんじほう    20

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

味方のポケモン

- 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

- 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

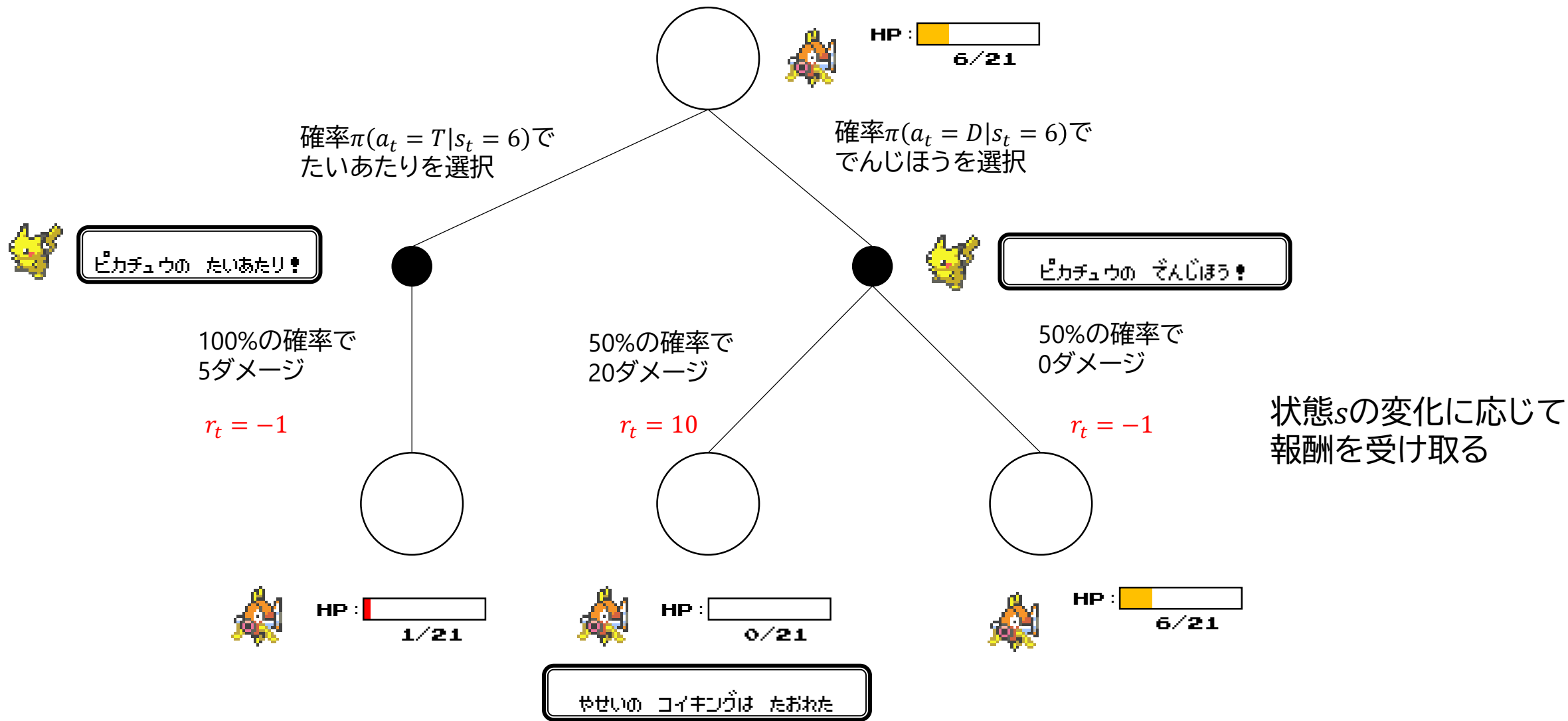
$$G_t = \sum_{i=1}^t r_i = r_1 + r_2 + \cdots + r_t$$

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



# バックアップ線図



# 状態価値関数の比較(改善した方策も含む)

10

- 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



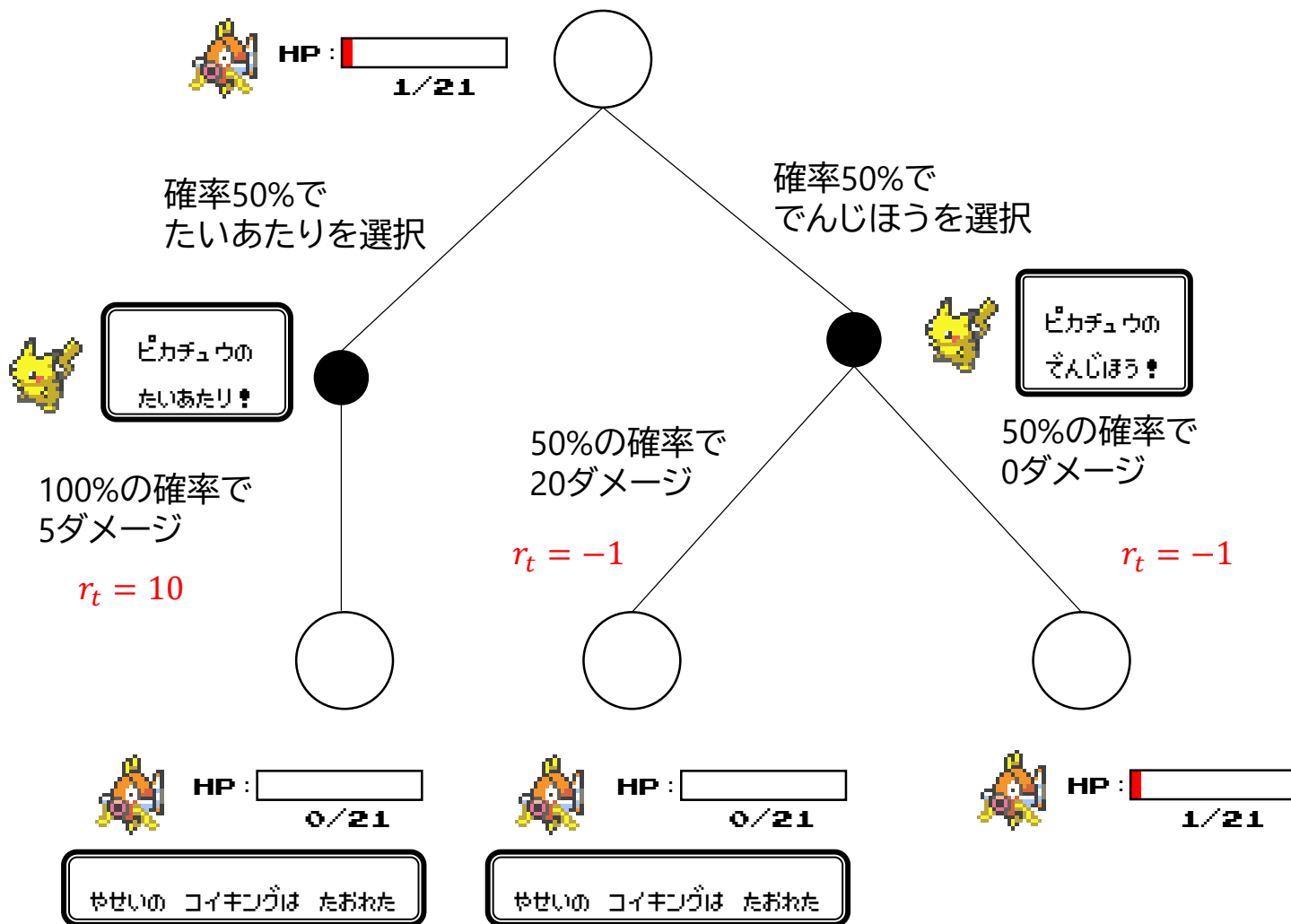
## 2. 価値関数と最適方策を推定する (動的計画法)

- 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

12

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1のとき(左のバックアップ線図)

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

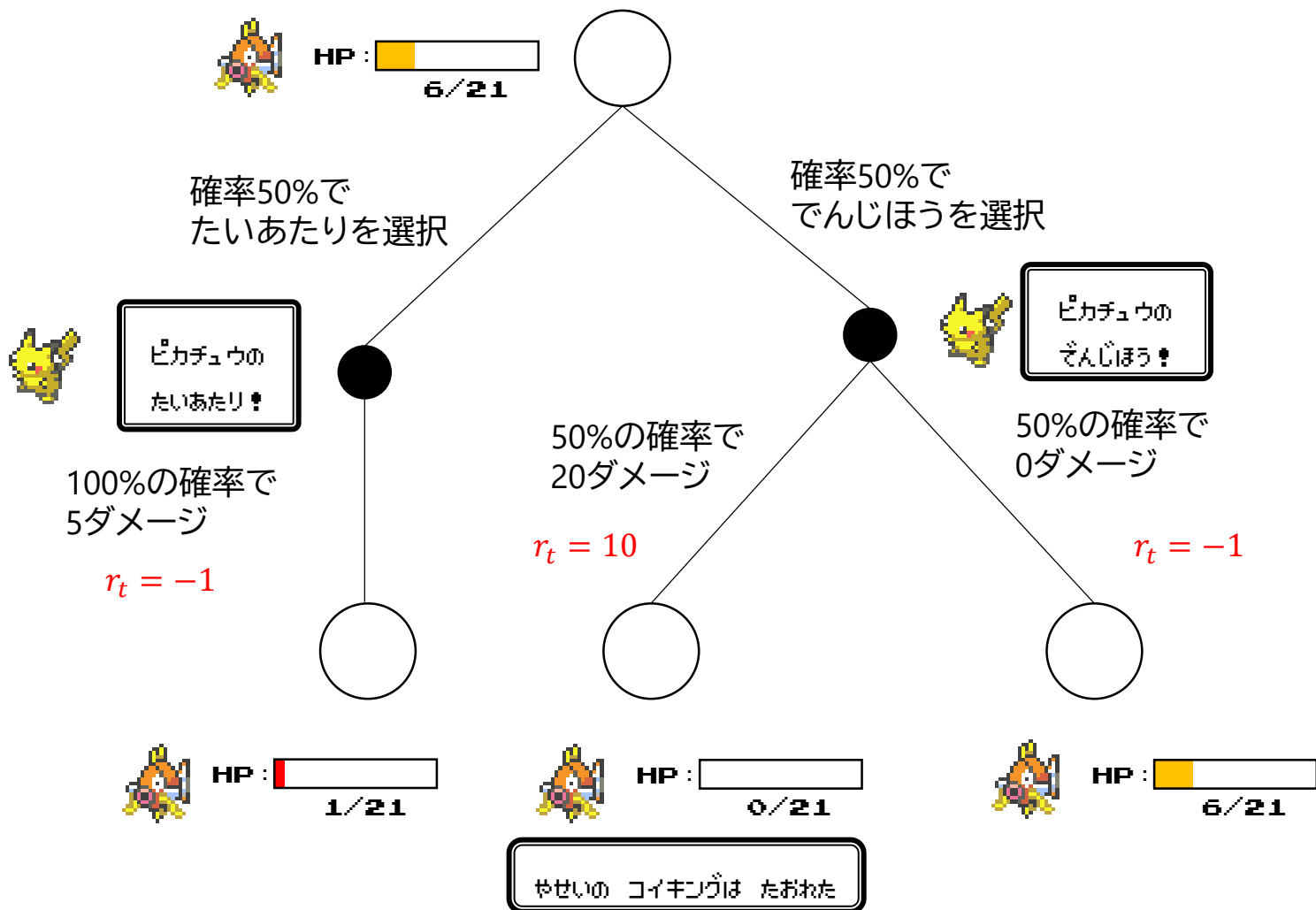
$v_R(0) = 0$ を代入して解くと

$$v_R(1) = \frac{29}{3} = 9.66 \dots$$

# ベルマン方程式を解いてみる③

13

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

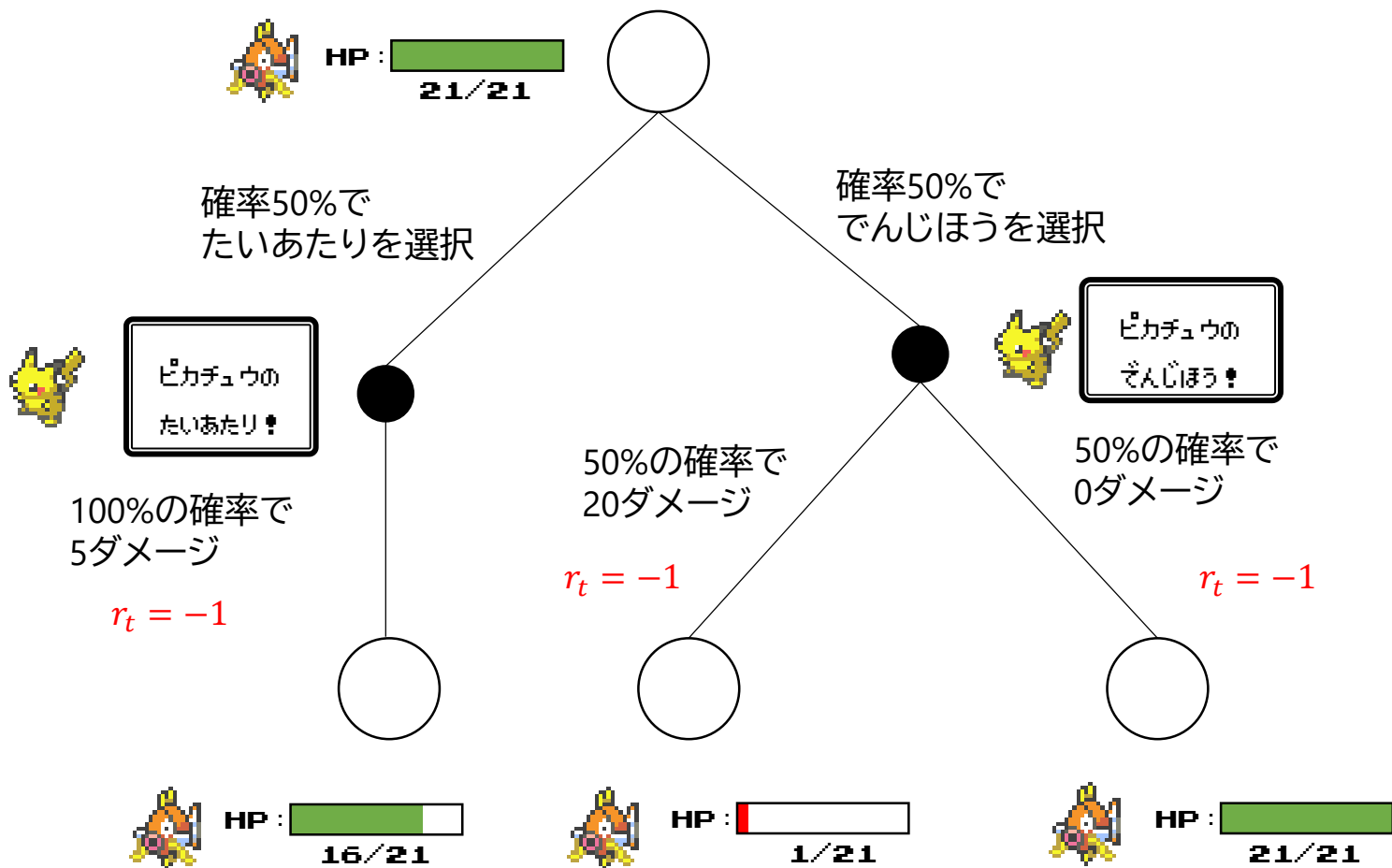
$$v_R(11) = \frac{7 + 2v_R(6)}{3} = \frac{221}{27} = 8.18 \dots$$

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

# ベルマン方程式を解いてみる③

14

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  
(コイキングの最大HPや、技の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