Solving the Dice Game Pig: an introduction to dynamic programming and value iteration

Todd W. Neller*, Ingrid Russell, Zdravko Markov

July 5, 2005

1 The Game of Pig

The object of the jeopardy dice game Pig is to be the first player to reach 100 points. Each player's turn consists of repeatedly rolling a die. After each roll, the player is faced with two choices: *roll* again, or *hold* (decline to roll again).

- If the player rolls a 1, the player scores nothing and it becomes the opponent's turn.
- If the player rolls a number other than 1, the number is added to the player's *turn total*, the sum of the rolls during the turn, and the player's turn continues.
- If the player holds, the turn total is added to the player's score, and it becomes the opponent's turn.

For such a simple dice game, one might expect a simple optimal strategy, such as in Blackjack (e.g., "stand on 17" under certain circumstances, etc.). As we shall see, this simple dice game yields a much more complex and intriguing optimal policy.

In our exploration of Pig we will learn about dynamic programming and value iteration, covering fundamental concepts of reinforcement learning techniques.

For the interested reader, there is a companion Game of Pig website[1] that features an optimal Pig computer player, VRML visualizations of the optimal policy, and information about Pig and its variants.

1.1 Simple Tactics

The game of Pig is simple to describe, but is it simple to play well? More specifically, how can we play the game optimally? Knizia [5] describes simple tactics where each roll is viewed as a bet that a 1 will not be rolled:

> ... we know that the true odds of such a bet are 1 to 5. If you ask yourself how much you should risk, you need to know how much there is to gain. A successful throw produces one of the numbers 2, 3, 4, 5, and 6. On average, you will gain four points. If you put 20 points at stake this brings the odds to 4 to 20, that is 1 to 5, and makes a fair game. ... Whenever your accumulated points are less than 20, you should continue throwing, because the odds are in your favor.
>
> [5, p. 129]

*Corresponding author: tneller@gettysburg.edu, Gettysburg College, Department of Computer Science, Campus Box 402, Gettysburg, PA 17325-1486

[1] See URL `http://cs.gettysburg.edu/projects/pig/index.html`.

However, Knizia also notes that there are many circumstances in which one should deviate from this "hold at 20" policy. Why does this reasoning not dictate an optimal policy for all play? The reason is that

risking points is not the same as risking the probability of winning.

Put another way, playing to maximize expected score is different from playing to win. For a clear illustration, consider the following extreme example. Your opponent has a score of 99 and will likely win in the next turn. You have a score of 78 and a turn total of 20. Do you follow the "hold at 20" policy and end your turn with a score of 98? Why not? Because the probability of winning if you roll once more is higher than the probability of winning if the other player is allowed to roll.

The "hold at 20" policy may be a good rule of thumb, but how good is it? Under what circumstances should we deviate from it and by how much?

1.2 Maximizing the Probability of Winning

Let $P_{i,j,k}$ be the player's probability of winning if the player's score is i, the opponent's score is j, and the player's turn total is k. In the case where $i + k \geq 100$, we have $P_{i,j,k} = 1$ because the player can simply hold and win. In the general case where $0 \leq i, j < 100$ and $k < 100 - i$, the probability of a player who plays optimally (an *optimal player*) winning is

$$P_{i,j,k} = \max\left(P_{i,j,k,\text{roll}}, P_{i,j,k,\text{hold}}\right),$$

where $P_{i,j,k,\text{roll}}$ and $P_{i,j,k,\text{hold}}$ are the probabilities of winning for rolling or holding, respectively. That is, the optimal player will always choose the action yielding the higher probability of winning. These probabilities are

$$P_{i,j,k,\text{roll}} = \frac{1}{6}((1 - P_{j,i,0}) + P_{i,j,k+2} + P_{i,j,k+3} + P_{i,j,k+4} + P_{i,j,k+5} + P_{i,j,k+6})$$

$$P_{i,j,k,\text{hold}} = 1 - P_{j,i+k,0}$$

The probability of winning after rolling a 1 or after holding is the probability that the other player will not win beginning with the next turn. All other outcomes are positive and dependent on the probabilities of winning with higher turn totals.

At this point, we can see how to compute the optimal policy for play. If we can solve for all probabilities of winning in all possible game states, we need only compare $P_{i,j,k,\text{roll}}$ with $P_{i,j,k,\text{hold}}$ for our current state and either roll or hold depending on which has a higher probability of resulting in a win.

Solving for the probability of a win in all states is not trivial, as dependencies between variables are cyclic. For example, $P_{i,j,0}$ depends on $P_{j,i,0}$ which in turn depends on $P_{i,j,0}$. This feature is easily illustrated when both players roll a 1 in subsequent turns. Put another way, game states can repeat, so we cannot simply evaluate probabilities from the end of the game backwards to the beginning, as in dynamic programming or its game-theoretic form, known as the *minimax process* (introduced in [12]; for a modern introduction to that subject, see [10, Ch. 6]).

1.3 Exercises

1. **Solving by hand:** Solve for the following win probabilities. Each set is dependent on the solution of the previous set

   (a) $P_{99,99,0}$,

   (b) $P_{98,99,0}$ and $P_{99,98,0}$, and

(c) $P_{97,99,0}$, $P_{97,99,2}$, and $P_{99,97,0}$.

2. **The Gambler's Fallacy:** In a *Mathematics Teaching in the Middle School* article [3], a variant of Pig called SKUNK was featured as part of an engaging middle-school exercise that encourages students to think about chance, choice, and strategy. The article states, "To get a better score, it would be useful to know, on average, how many good rolls happen in a row before a 'one' or 'double ones' come up." (In this Pig variant, a 'one' or 'double ones' are undesirable rolls.) Define "the Gambler's Fallacy" and explain why this statement is an example of a Gambler's Fallacy.

2 Dynamic Programming

Dynamic programming is a powerful technique which uses memory to reduce redundant computation. Although dynamic programming is not directly applicable to the solution of Pig, we will see that it can approximate the optimal policy through application to a very similar game.

We will first introduce dynamic programming through the simple task of computing Fibonacci sequence numbers. Then we will introduce and solve Progressive Pig, a slight variation of Pig.

2.1 Remembering Fibonacci Numbers

As a simple illustrative example, consider the Fibonacci sequence $1, 1, 2, 3, 5, 8, 13 \ldots$ defined as follows:

$$\text{fib}_n = \begin{cases} 1 & n = 1, 2 \\ \text{fib}_{n-1} + \text{fib}_{n-2} & n > 2 \end{cases}$$

2.1.1 Simple Recursive Approach

From this recursive definition, we can write a simple recursive algorithm to compute `fib(n)` for some positive integer `n`:

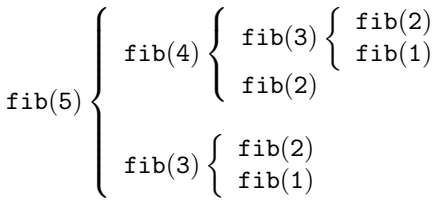
⟨*Compute fib(n)*⟩≡
```
public static long fib(int n) {
    if (n <= 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Side note: This format of presenting code in chunks is called *literate programming* and is due to Donald Knuth. Code will be presented in named *chunks* that will appear inserted within other chunk definitions. The source document is used not only to generate the output you are now reading, but also to generate the example code. In this way, the code presented is both consistent with the example code, and has been tested for correctness.

Observe the behavior of this algorithm as we test it for successively higher values of `n`. (You will need to stop the process.)

⟨*Test recursive implementation*⟩≡
```
for (int n = 1; n < MAX; n++)
    System.out.println("fib(" + n + ") = " + fib(n));
```

The reason that the process slows considerably is that the number of recursive calls increases exponentially with `n`. Consider a call to `fib(5)`. This in turn causes recursive calls to `fib(4)` and `fib(3)`. These in turn have recursive calls of their own, illustrated below:



Note that the computation for `fib(3)` is performed twice. This problem of redundant computation becomes even more noticeable as `n` increases. The following table shows how many times each recursive call is performed as `n` increases[2].

	Recursive Calls for `fib(n)`				
`n`	`fib(1)`	`fib(2)`	`fib(3)`	`fib(4)`	`fib(5)`
1	1	0	0	0	0
2	0	1	0	0	0
3	1	1	1	0	0
4	1	2	1	1	0
5	2	3	2	1	1
6	3	5	3	2	1
7	5	8	5	3	2
8	8	13	8	5	3
9	13	21	13	8	5
10	21	34	21	13	8
...					
20	2584	4181	2584	1597	987
...					
30	317811	514229	317811	196418	121393
...					
40	39088169	63245986	39088169	24157817	14930352

2.1.2 Dynamic Programming Approach

We can avoid this computational time inefficiency by storing the Fibonacci numbers that have been computed and retrieving them as needed.

[2]It is interesting to note that the Fibonacci sequence appears in each table column, with the ratio of successive pairs asymptotically approaching the *golden ratio* of $(1 + \sqrt{5})/2$.

⟨*Compute fib(n) with dynamic programming*⟩≡
```
public static final int MAX = 90;
public static boolean[] computed = new boolean[MAX];
public static long[] result = new long[MAX];

public static long fibDP(int n) {
    // Compute and store value if not already stored
    if (!computed[n]) {
        if (n <= 2)
            result[n] = 1;
        else
            result[n] = fibDP(n - 1) + fibDP(n - 2);
        computed[n] = true;
    }

    // Retrieve and return stored value
    return result[n];
}
```

Now observe the behavior of this dynamic programming algorithm as we test it for successively higher values of `n`.

⟨*Test recursive implementation with dynamic programming*⟩≡
```
for (int n = 1; n < MAX; n++)
    System.out.println("fibDP(" + n + ") = " + fibDP(n));
```

The full test code implementation which computes using dynamic programming first is given as follows:

⟨*FibDemo.java*⟩≡
```
public class FibDemo {

    ⟨Compute fib(n) with dynamic programming⟩

    ⟨Compute fib(n)⟩

    public static void main(String[] args) {
        ⟨Test recursive implementation with dynamic programming⟩
        ⟨Test recursive implementation⟩
    }
}
```

The key tradeoff to observe between these algorithms is that dynamic programming uses additional memory to cut computational time complexity from exponential to linear[3]. Memory is used to save time. This is a common tradeoff in the art of algorithm design. Given the relative cheapness of memory there are many problems where it makes sense to store the results of computations to avoid recomputing them.

2.2 Progressive Pig

Now we turn our attention back to Pig. We cannot simply solve Pig using a recursive approach with dynamic programming, because there are cyclic dependencies between the variables. Dynamic pro-

[3]Of course, there is a simpler linear time algorithm for computing Fibonacci numbers. However, many problems are based on a complex set of interacting subproblems not amenable to such an approach.

gramming depends on acyclic dependencies between computations that allow us to compute results sequentially from simple (base) computations without such dependencies, to those computations which depend on the base computations, etc. In the Figure 1, we visualize computations as nodes. Dependencies of computations on the results of other computations are indicated by arrows. Each stage of dynamic programming computation (concentric ellipses) is dependent only upon the computations of previous stages. The cyclic dependencies of Pig (e.g. $P_{i,j,0} \rightarrow P_{j,i,0} \rightarrow P_{i,j,0}$) prevent us from dividing its computation into such stages.

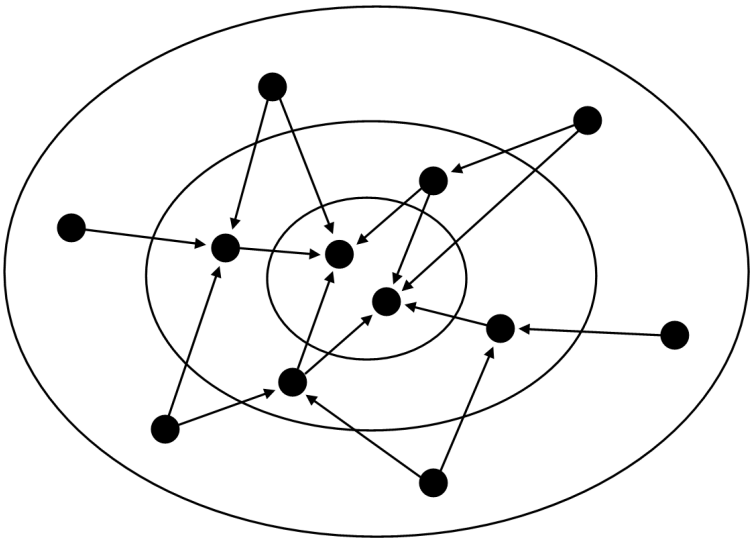


Figure 1: Partitioning dynamic programming computation into stages

However, we can approximate optimal play for Pig by making a small change to the rules that makes the variable dependencies acyclic. That is, we slightly modify the game such that game states can never repeat and always progress towards the end of the game. We will call this modified game *Progressive Pig*. Optimal play for Progressive Pig will approximate optimal play for Pig.

Progressive Pig is identical to Pig except that a player always scores at least 1 point each turn:

- If the player rolls a 1, the player scores 1 point and it becomes the opponent's turn.
- If the player rolls a number other than 1, the number is added to the player's turn total and the player's turn continues.
- If the player holds, the greater of 1 and the turn total is added to the player's score and it becomes the opponent's turn.

Thus the equations for $P'_{i,j,k} = \max(P'_{i,j,k,\text{roll}}, P'_{i,j,k,\text{hold}})$, the probability of winning Progressive Pig with optimal play, are

$$P'_{i,j,k,\text{roll}} = \frac{1}{6}((1 - P'_{j,i+1,0}) + P'_{i,j,k+2} + P'_{i,j,k+3} + P'_{i,j,k+4} + P'_{i,j,k+5} + P'_{i,j,k+6})$$

$$P'_{i,j,k,\text{hold}} = 1 - P'_{j,i+\max(k,1),0}$$

2.3 Solving Progressive Pig

To solve Progressive Pig (abbreviated P-Pig), we keep track of the `goal` score, and establish variables to manage and store the results of our computations. The 3D boolean array `computed` keeps track of which (i, j, k) states have been computed. The computed value `p[i][j][k]` corresponds to $P'_{i,j,k}$. The computed value `roll[i][j][k]` indicates whether or not it is optimal to roll in state (i, j, k).

⟨*P-Pig variable definitions*⟩≡
```
int goal;
boolean[][][] computed;
double[][][] p;
boolean[][][] roll;
```

When constructing a solution to P-Pig, we supply the `goal` score as a parameter.

⟨*Solve P-Pig*⟩≡
```
PPigSolver(int goal) {
    this.goal = goal;
    computed = new boolean[goal][goal][goal];
    p = new double[goal][goal][goal];
    roll = new boolean[goal][goal][goal];
    ⟨Compute all win probabilities⟩
}
```

After we initialize the variables, we compute all win probabilities for all states.

⟨*Compute all win probabilities*⟩≡
```
for (int i = 0; i < goal; i++) // for all i
    for (int j = 0; j < goal; j++) // for all j
        for (int k = 0; i + k < goal; k++) // for all k
            pWin(i, j, k);
```

In method `pWin` below, we first check to see if one player has won, returning a win probability of 0 or 1 depending on which player reached 100. Note the implicit assumption that an optimal player with a winning turn total will hold and win. This limits us to a finite state space. Secondly, we check to see if this probability has already been computed and stored. If so, we return it. This is our dynamic programming step. Then, if we have not yet returned a result, we must compute it. However, these previous steps ensure that we will not be redundantly computing probabilities in the recursive calls.

⟨*Compute the probability of winning with optimal play*⟩≡
```
public double pWin(int i, int j, int k) {
    if (i + k >= goal) return 1.0;
    if (j >= goal) return 0.0;
    if (computed[i][j][k]) return p[i][j][k];

    ⟨Recursively compute p[i][j][k]⟩
    return p[i][j][k];
}
```

To recursively compute `p[i][j][k]`, we merely translate the equations of Section 2.2 to code. Below, `pRoll` and `pHold` represent the probabilities of winning with a roll and a hold, respectively.

⟨*Recursively compute p[i][j][k]*⟩≡
```
// Compute the probability of winning with a roll
double pRoll = 1.0 - pWin(j, i + 1, 0);
for (int roll = 2; roll <= 6; roll++)
    pRoll += pWin(i, j, k + roll);
pRoll /= 6.0;

// Compute the probability of winning with a hold
double pHold;
if (k == 0)
    pHold = 1.0 - pWin(j, i + 1, 0);
else
    pHold = 1.0 - pWin(j, i + k, 0);

// Optimal play chooses the action with the greater win probability
roll[i][j][k] = pRoll > pHold;
if (roll[i][j][k])
    p[i][j][k] = pRoll;
else
    p[i][j][k] = pHold;
computed[i][j][k] = true;
```

We now include code to summarize results of the computation. We first print the probability of a first player win with optimal play. Then for each i, j pair, we list the k values where the player changes policy (e.g. from roll to hold).

⟨*Summarize results*⟩≡
```
public void summarize() {
    System.out.println("p[0][0][0] = " + p[0][0][0]);
    System.out.println();
    System.out.println("i\tj\tPolicy changes at k =");
    for (int i = 0; i < goal; i++) // for all i
        for (int j = 0; j < goal; j++) { // for all j
            int k = 0;
            System.out.print(i + "\t" + j + "\t" + ⟨Policy for (i,j,k)⟩);
            for (k = 1; i + k < goal; k++) // for all valid k
                if (roll[i][j][k] != roll[i][j][k-1])
                    System.out.print(k + " " + ⟨Policy for (i,j,k)⟩);
            System.out.println();
        }
}
```

Where the policy string "hold" or "roll" is chosen using the Java selection operator:

⟨*Policy for (i,j,k)*⟩≡
```
(roll[i][j][k] ? "roll " : "hold ")
```

The output line "15 70 roll 25 hold 35 roll 38 hold 77 roll" would indicate that when a player has a score of 15 their opponent has 70, the player should roll for turn total values 0-24, 35-37, and ≥ 77. Note that, in practice, an optimal player would never reach a turn total of 35 points, as there is is no way to pass from 0-24 to 35 without passing through a "hold" state.

Finally, we put these pieces together and test the computation with the goal set to 100.

⟨*PPigSolver.java*⟩≡

```
public class PPigSolver {
    ⟨P-Pig variable definitions⟩
    ⟨Solve P-Pig⟩
    ⟨Compute the probability of winning with optimal play⟩
    ⟨Summarize results⟩

    public static void main(String[] args) {
        new PPigSolver(100).summarize();
    }
}
```

2.4 Exercises

3. **Pig Solitaire:** Consider the solitaire (single player) game of Pig where a player is challenged to reach a given goal score g within n turns.

   (a) Define the state space.

   (b) Write the equations that describe optimal play.

   (c) Prove that the state space is acyclic, i.e. that states cannot repeat.

   (d) Compute the optimal policy for $g = 100$ and $n = 10$. Again, assume that an optimal player with a winning turn total will hold and win.

   (e) Summarize or visualize the policy, and describe it qualitatively in your own words.

   (f) For $g = 100$, what is the smallest n for which the optimal player's initial win probability is $\geq .50$?

4. **Pig Solitaire 2:** Consider the solitaire (single player) game of Pig where one is challenged to maximize one's score within n turns. Now, rather than seeking to maximize the probability of a win, one seeks to maximize the expected score.

   (a) Define the state space.

   (b) Write the equations that describe optimal play.

   (c) Prove that the state space is acyclic, i.e. that states cannot repeat.

   (d) Compute the optimal policy for $n = 5$. In order to limit ourselves to a finite state space, assume that the player will always hold with a sufficiently high score or turn total (e.g. $i, k \geq 500$). You will need to experiment with different limits to be assured that your policy is optimal and not affected by your limits.

   (e) Summarize or visualize the policy, and describe it qualitatively in your own words.

5. **THINK Solitaire:** In [4], Falk and Tadmor-Troyanski analyze a 2-dice Pig variant called THINK. THINK is identical to Pig, except that

   - Two standard dice are rolled. If neither shows a 1, their sum is added to the turn total.
   - If a single 1 is rolled, the player's turn ends with the loss of the turn total.
   - If two 1's are rolled, the player's turn ends with the loss of the turn total *and score*.
   - Each player gets only five turns, one for each letter of THINK.
   - The highest score at the end of five turns wins.

In this exercise, you will compute optimal play for a solitaire player seeking to maximize their THINK score in five turns.

(a) Define the state space.

(b) Write the equations that describe optimal play.

(c) Prove that the state space is acyclic, i.e. that states cannot repeat.

(d) Compute the optimal policy. In order to limit ourselves to a finite state space, assume that the player will always hold with a sufficiently high score or turn total (e.g. $i, k \geq 500$). You will need to experiment with different limits to be assured that your policy is optimal and not affected by your limits.

(e) Summarize or visualize the policy, and describe it qualitatively in your own words.

2.5 Advanced Projects

2.5.1 Risk

The board game Risk®, first published in 1959, is arguably the most popular war game internationally. Players seek global domination through a series of battles between adjacent territories on a simplified world map.

The outcome of conflicts between territories occupied by army pieces are determined by dice rolling. The attacker declares the intent of rolling 1, 2, or 3 dice. The attacker must have at least one more army than the number of dice rolled. The defender then declares the intent of rolling 1 or 2 dice. The defender must have at least as many armies as the number of dice rolled. Both players roll their declared number of dice and sort them. Highest dice and, if applicable, second highest dice of the players are compared. For each pair the player with the lower number loses an army. If the pair is tied, the attacker loses an army. For example, suppose the attacker rolls 5-3-1 and the defender rolls 4-3. Comparing 5 and 4, the defender removes an army. Comparing 3 and 3, the attacker removes an army. After each roll, the attacker decides whether to retreat or continue the attack, repeating the process. Risk rules[4] may be found at Hasbro's website.

Several probabilistic analyses of Risk battles have been published in recent years. In [9], Jason Osborne of North Carolina State University computed odds of victory in a Risk battle under the assumption that

- the attacker never retreats, pressing the attack until victorious or reduced to 1 army, and
- both players always roll the maximum permissible number of dice.

Confirm the results of [9][5].

As a more advanced exercise, devise and compute a more advanced Risk analysis. For example, one Risk tactic is to eliminate a player, claiming the defeated player's valuable Risk cards. Although one can often perceive a chain of attacks that might achieve this goal, it is difficult to assess the probable outcome of such a series of attacks. After confirming the results of [9], consider generalizing your work to answer the following question:

> Given a positive number of attacking armies a, and a sequence $d_1, \ldots, d_n$ of the positive number of defending armies in successively adjacent territories,
>
> - What is the probability of victory, i.e. total occupation of the chain of territories?
> - How many armies should the attacker expect to lose on average?

[4] See URL `http://www.hasbro.com/common/instruct/Risk1999.PDF`.

[5] already confirmed by the author

- How many armies should the defender expect to lose on average?

Assume that the attacker never retreats, that both players always roll the maximum permissible number of dice, and that the attacker always moves all but one army into a conquered territory.

The author has computed such probabilities and confirmed interesting properties of defensive configurations. Given such a program, the author recommends the following puzzle:

> In a chain of 6 territories, suppose an attacker occupies the leftmost territory with 30 armies, and the defender occupies the remaining 5 territories with 30 armies. How should the defender distribute these armies so as to minimize the attacker's probability of successful chain occupation? (Each territory must contain at least one army.)

It is often advantageous to maintain a strong front in the game. That is, ones armies should usually be concentrated in those territories adjacent to opponent territories. Compute win probabilities for configurations with 30 attackers and 30 defenders distributed in chains as follows:

- 26, 1, 1, 1, 1
- 22, 2, 2, 2, 2
- 18, 3, 3, 3, 3
- 14, 4, 4, 4, 4
- 10, 5, 5, 5, 5
- 6, 6, 6, 6, 6

What do you observe? Compare your observations with those of section 3.3 of the Risk FAQ[6].

2.5.2 Yahtzee

In each turn of the popular dice game Yahtzee®, players can roll and reroll dice up to three times each turn in order to form a high-scoring combination in one of several scoring categories. Hasbro's Yahtzee rules[7] can be found online.

Phil Woodward has computed optimal solitaire play for Yahtzee, i.e. the policy that maximizes score for a single player [13]. Although Yahtzee can in principle be solved for any number of players with dynamic programming, the size of the state space as well as details such as the bonus rules make this a challenging project for the basic solitaire case.

For students wishing to compute optimal play for a simpler solitaire game similar to Yahtzee, there are a number of Yahtzee variants described in [5]. For example, the category dice game of Hooligan allows scoring in seven categories: six number categories (Ones, Twos, Threes, Fours, Fives, and Sixes), and Hooligan (a straight: 1-2-3-4-5 or 2-3-4-5-6). Rules for Hooligan can also be found online at Dice-Play's Hooligan Dice Game page[8]. You may even wish to invent your own simplified variant of Yahtzee.

Optimal solitaire play for even the simplest variants may surprise you!

[6] See URL `http://www.kent.ac.uk/IMS/personal/odl/riskfaq.htm`.
[7] See URL `http://www.hasbro.com/common/instruct/Yahtzee.pdf`.
[8] See URL `http://homepage.ntlworld.com/dice-play/Games/Hooligan.htm`.

3 Value Iteration

Value iteration [11, 1, 2] is a process by which we iteratively improve estimates of the value of being in each state until our estimates are "good enough." For ease of explanation, we will first introduce a simpler game we have devised called "Piglet." We will then describe value iteration and show how it is applied to Piglet.

3.1 Piglet

Piglet is very much like Pig except it is played with a coin rather than a die. The object of Piglet is to be the first player to reach 10 points. Each turn, a player repeatedly flips a coin until either a "tail" is flipped or the player holds and scores the number of consecutive "heads" flipped. At any time during the player's turn, the player is faced with two choices: *flip* or *hold*. If the coin turns up tails, the player scores nothing and it becomes the opponent's turn. Otherwise, the player's turn continues. If the player chooses to *hold*, the number of consecutively flipped heads is added to the player's score and it becomes the opponent's turn.

The number of equations necessary to express the probability of winning in each state is still too many for a pencil and paper exercise, so we will simplify this game further. Now suppose the object is to be the first player to reach 2 points.

As before, let $P''_{i,j,k}$ be the player's probability of winning if the player's score is i, the opponent's score is j, and the player's turn total is k. In the case where $i + k = 2$, $P''_{i,j,k} = 1$ because the player can simply hold and win. In the general case where $0 \leq i, j < 2$ and $k < 2 - i$, the probability of a player winning is

$$P''_{i,j,k} = \max\left(P''_{i,j,k,\text{flip}}, P''_{i,j,k,\text{hold}}\right)$$

where $P''_{i,j,k,\text{flip}}$ and $P''_{i,j,k,\text{hold}}$ are the probabilities of winning if one flips and holds, respectively. The probability of winning if one flips is

$$P''_{i,j,k,\text{flip}} = .5((1 - P''_{j,i,0}) + P''_{i,j,k+1})$$

The probability $P''_{i,j,k,\text{hold}}$ is just as before. Then the equations for the probabilities of winning in each state are given as follows:

$$\begin{aligned}
P''_{0,0,0} &= \max\left(.5((1 - P''_{0,0,0}) + P''_{0,0,1}), 1 - P''_{0,0,0}\right) \\
P''_{0,0,1} &= \max\left(.5((1 - P''_{0,0,0}) + 1), 1 - P''_{0,1,0}\right) \\
P''_{0,1,0} &= \max\left(.5((1 - P''_{1,0,0}) + P''_{0,1,1}), 1 - P''_{1,0,0}\right) \\
P''_{0,1,1} &= \max\left(.5((1 - P''_{1,0,0}) + 1), 1 - P''_{1,1,0}\right) \\
P''_{1,0,0} &= \max\left(.5((1 - P''_{0,1,0}) + 1), 1 - P''_{0,1,0}\right) \\
P''_{1,1,0} &= \max\left(.5((1 - P''_{1,1,0}) + 1), 1 - P''_{1,1,0}\right)
\end{aligned} \tag{1}$$

Once these equations are solved, the optimal policy is obtained by observing which action maximizes $\max\left(P''_{i,j,k,\text{flip}}, P''_{i,j,k,\text{hold}}\right)$ for each state.

3.2 Value Iteration

Value iteration is an algorithm that iteratively improves estimates of the value of being in each state. In describing value iteration, we follow [11], which we also recommend for further reading. We assume that the world consists of *states*, *actions*, and *rewards*. The goal is to compute which action to take in each state so as to maximize future rewards. At any time, we are in a known state s of a finite set of states $\mathcal{S}$. For each state s, there is a finite set of allowable actions $\mathcal{A}_f$. For any two states $s, s' \in \mathcal{S}$

and any action $a \in \mathcal{A}_f$, there is a probability $\mathcal{P}^a_{ss'}$ (possibly zero) that taking action a will cause a transition to state s'. For each such transition, there is an expected immediate *reward* $\mathcal{R}^a_{ss'}$.

We are not just interested in the immediate rewards; we are also interested to some extent in future rewards. More specifically, the *value* of an action's result is the sum of the immediate reward plus some fraction of the future reward. The *discount factor* $0 \leq \gamma \leq 1$ determines how much we care about expected future reward when selecting an action.

Let $V(s)$ denote the estimated value of being in state s, based on the expected immediate rewards of actions *and* the estimated values of being in subsequent states. The estimated value of an action a in state s is given by:

$$\sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$$

Since any action can be chosen, the optimal choice is the action that maximizes this estimated value:

$$\max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$$

This expression serves as an estimate of the value of being in state s, that is, $V(s)$. In a nutshell, value iteration consists of revising the estimated values of states until they converge, i.e., until no single estimate is changed significantly. The algorithm is given as Algorithm 1.

Algorithm 1 Value iteration

For each $s \in \mathcal{S}$, initialize $V(s)$ arbitrarily.
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$,
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \epsilon$

Algorithm 1 repeatedly updates estimates of $V(s)$ for each s. The variable Δ is used to keep track of the largest change for each iteration, and ϵ is a small constant. When the largest estimate change Δ is smaller than ϵ, we stop revising our estimates.

In general, convergence is not guaranteed when $\gamma = 1$. In fact, convergence is guaranteed only when $\gamma < 1$ and rewards are bounded [7, §13.4]. In the cases of Piglet and Pig, value iteration happens to converge when $\gamma = 1$.

3.3 Applying Value Iteration to Piglet

Value iteration is beautiful in its simplicity, but we have yet to show how it applies to Piglet. For Piglet with a goal of 2, we denote states $s_{i,j,k}$ for all possible (i, j, k) triples that can occur in game play, where i, j and k denote the same game values as before. Additionally, we have a *terminal state* s_∞ representing all situations in which a player has won or lost. All actions taken in s_∞ transition to s_∞, causing no change and yielding no reward. The set of actions for all states is $\mathcal{A} = \{\text{flip}, \text{hold}\}$.

Let us consider rewards carefully. If points are our reward, then we are once again seeking to maximize expected points rather than maximizing the expected probability of winning. Instead, in order to only reward winning, we have a reward of 1 for *winning* transitions from $s_{i,j,k}$ to s_∞. All other rewards are set to 0.

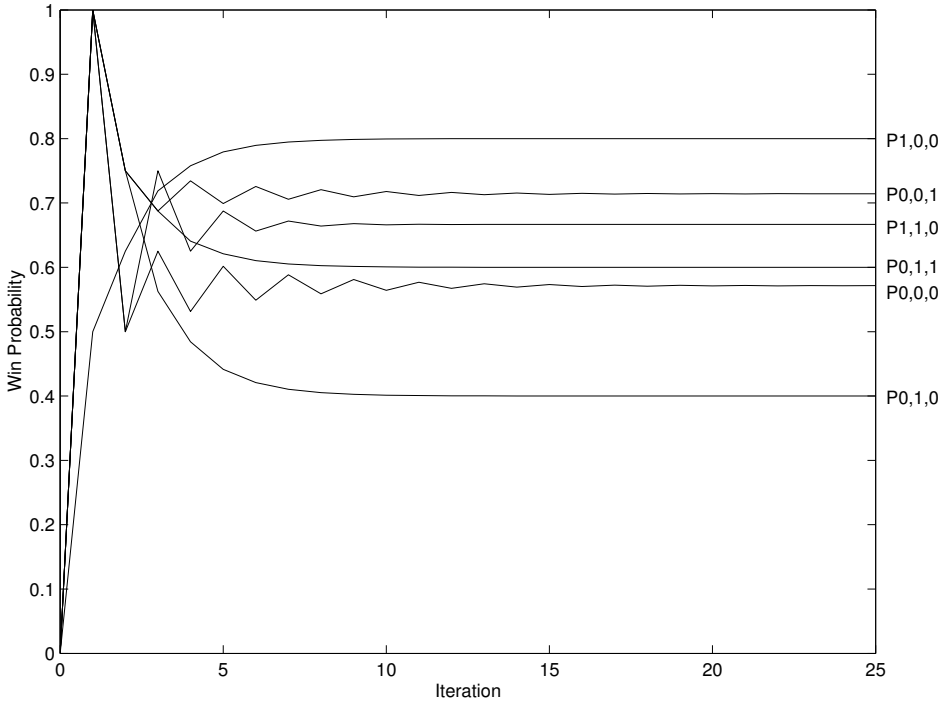


Figure 2: Value Iteration with Piglet (goal points = 2).

The next bit of insight that is necessary concerns what happens when we offer a reward of 1 for winning and do *not* discount future rewards, that is, when we set $\gamma = 1$. In this special case, $V(s)$ is the probability of a player in s eventually winning. That is, $V(s_{i,j,k}) = P''_{i,j,k}$ and $V(s_\infty) = 0$.

The last insight we need is to note the symmetry of the game. Each player has the same choices and the same probable outcomes. It is this fact that enables us to use $(1 - P''_{j,i,0})$ and $(1 - P''_{j,i+k,0})$ in our Pig/Piglet equations. Thus, we only need to consider the perspective of a single optimal player. [9]

Now when we review our system of equations for Piglet, we see that value iteration with $\gamma = 1$ amounts to computing the system's left-hand side probabilities (e.g., $P''_{i,j,k}$) from the right-hand side expressions (e.g., $\max(P''_{i,j,k,\text{flip}}, P''_{i,j,k,\text{hold}})$) repeatedly until the probabilities converge.

The result of applying value iteration to Piglet is shown in Figure 2. Each line corresponds to a sequence of estimates made for one of the win probabilities for our Piglet equations. The interested reader can verify that the exact values, $P''_{0,0,0} = \frac{4}{7}, P''_{0,0,1} = \frac{5}{7}, P''_{0,1,0} = \frac{2}{5}, P''_{0,1,1} = \frac{3}{5}, P''_{1,0,0} = \frac{4}{5}, P''_{1,1,0} = \frac{2}{3}$, do indeed solve the system of equations (1).

Finally, we note that the optimal policy for play is simply computed by observing which action yields the maximum expected value for each state. In the case of Piglet with a goal of 2, one should always keep flipping. Piglet with a goal of 10 has a more interesting optimal policy, and Pig, with different possible positive outcomes for rolling, has an optimal policy that is more interesting still.

3.4 Solving Piglet

In this section, we apply value iteration to the solution of Piglet, explaining each part of the solution in detail.

[9] If we wish to treat the multi-agent case, then we are dealing with a simple type of *Markov game* [6] with non-simultaneous actions, and can apply multi-agent generalizations of reinforcement learning techniques.

3.4.1 Preliminaries

To solve Piglet, we keep track of the `goal` score, the convergence parameter `epsilon` for value iteration, the current estimate `p` of the probability of a win playing optimally from each possible game state.

⟨*Variable definitions*⟩≡
```
int goal;
double epsilon;
double[][][] p;
boolean[][][] flip;
```

When constructing a solution to Piglet, we supply as parameters the `goal` score and the convergence parameter `epsilon` for value iteration. After we *Initialize variables*, we *Perform value iteration*.

⟨*Construct solution*⟩≡
```
PigletSolver(int goal, double epsilon) {
    ⟨Initialize variables⟩
    valueIterate();
}
```

In keeping track of win probability estimates, we are only concerned with game states where the game is not over. We assume that a player will hold if they have a turn total that is adequate to reach the `goal` score, that is, a turn total less than the `goal` score minus the player's score i. All win probability estimates are initially 0 by default.

⟨*Initialize variables*⟩≡
```
this.goal = goal;
this.epsilon = epsilon;
p = new double[goal][goal][goal];
flip = new boolean[goal][goal][goal];
```

3.4.2 Performing value iteration

In value iteration, we repeatedly recompute utility estimates until the greatest estimate change is very small. For this problem, our utility is the probability of winning. We first present the core value iteration algorithm, and then give specifics on how the utilities/probabilities are computed.

For each iteration of value iteration, we must recompute $P''_{i,j,k}$ for all i, j, k, noting the old probability estimate, and keeping track of the largest change of the iteration in local variable `maxChange` (Δ). We terminate value iteration when `maxChange` is less than our convergence parameter `epsilon` (ϵ).

⟨*Perform value iteration*⟩≡
```
void valueIterate() {
  double maxChange;
    do {
        maxChange = 0.0;
        for (int i = 0; i < goal; i++) // for all i
            for (int j = 0; j < goal; j++) // for all j
                for (int k = 0; k < goal - i; k++) { // for all k
                    double oldProb = p[i][j][k];
                    ⟨Compute new p[i][j][k] estimate⟩
                    double change = Math.abs(p[i][j][k] - oldProb);
                    maxChange = Math.max(maxChange, change);
                }
    } while (maxChange >= epsilon);
}
```

To recompute our probability estimates, it helps to have a function that allows us to look up current probability estimates without being concerned about array indices going out of bounds. For our equations, array indices are out of bounds when we consider terminal game states where either one player or another has enough points to win the game. In `pWin`, we check if the player or opponent have enough points to win, returning win probabilities of 1.0 and 0.0 respectively. Otherwise, we simply return our current estimate from `p`.

⟨*Return estimated probability of win*⟩≡
```
public double pWin(int i, int j, int k) {
    if (i + k >= goal)
        return 1.0;
    else if (j >= goal)
        return 0.0;
    else return p[i][j][k];
}
```

Using `pWin`, the equations of Section 1.2 yield the following computation:

⟨*Compute new p[i][j][k] estimate*⟩≡
```
double pFlip = (1.0 - pWin(j, i, 0) + pWin(i, j, k + 1)) / 2;
double pHold = 1.0 - pWin(j, i + k, 0);
p[i][j][k] = Math.max(pFlip, pHold);
flip[i][j][k] = pFlip > pHold;
```

3.4.3 Printing results

Once the solution is computed, we wish to summarize the results. Although the use of the `summarize` method of section 2.3 is recommended for the exercises, here we print a simple summarizing table of the turn totals at which a player should hold for each possible game situation. In each case, this is either (1) the winning turn total ($k = \text{goal} - i$), or the first k for which the optimal decision is to hold.

⟨*Output hold values*⟩≡
```
public void outputHoldValues() {
    for (int i = 0; i < goal; i++) {
        for (int j = 0; j < goal; j++) {
            int k = 0;
            while (k < goal - i && flip[i][j][k])
                k++;
            System.out.print(k + " ");
        }
        System.out.println();
    }
}
```

Finally, we construct a solution for Piglet with a `goal` score of 10 and a convergence `epsilon` of 10^{-9}, and then output the hold values.

⟨*Execute program*⟩≡
```
public static void main(String[] args){
    new PigletSolver(10, 1e-9).outputHoldValues();
}
```

Putting this all together, the program to solve Piglet is as follows:

⟨*PigletSolver.java*⟩≡
```
public class PigletSolver {
    ⟨Variable definitions⟩
    ⟨Construct solution⟩
    ⟨Perform value iteration⟩
    ⟨Return estimated probability of win⟩
    ⟨Output hold values⟩
    ⟨Execute program⟩
}
```

3.5 Exercises

6. **Pig:** In the preceding text, a solution is outlined for solving Piglet. Now, you will modify this approach to solve 2-player Pig with a goal score of 100.

   (a) Define the state space.

   (b) Write the equations that describe optimal play.

   (c) Compute the optimal policy. Assume that an optimal player with a winning turn total will hold and win. What is the probability that the first player will win if both players play optimally?

   (d) Summarize or visualize the policy, and describe it qualitatively in your own words.

7. **Pig Solitaire 3:** Consider the solitaire (single player) game of Pig where a player is challenged to minimize the turns taken to reach a given goal score g. Hint: Let the only reward be a reward of -1 at the end of each turn. In this way the value of the initial state will be the negated expected number of turns to reach the goal score g.

   (a) Define the state space.

(b) Write the equations that describe optimal play.

(c) Compute the optimal policy for $g = 100$. Assume that an optimal player with a winning turn total will hold and win. What is the expected number of turns to reach 100 when playing optimally?

(d) Summarize or visualize the policy, and describe it qualitatively in your own words.

8. **Pass the Pigs:** Pass the Pigs (a.k.a. Pigmania) is a popular commercial variant of Pig which involves rolling two rubber pigs to determine the change in turn total or score. Rules for Pass the Pigs can be found at the Hasbro website[10]. For simplicity, make the following assumptions:

   - Assume that the player can throw the pigs so as to make the probability of an "oinker" or "piggyback" effectively 0.
   - Assume probabilities for other pig rolls are accurately represented by the data at Freddie Wong's Pass the Pigs page[11]:

     Right Sider $\frac{1344}{3939}$
     Left (Dot) Sider $\frac{1294}{3939}$
     Razorback $\frac{767}{3939}$
     Trotter $\frac{365}{3939}$
     Snouter $\frac{137}{3939}$
     Leaning Jowler $\frac{32}{3939}$

   (a) Define the state space.

   (b) Write the equations that describe optimal play.

   (c) Compute the optimal policy. Assume that an optimal player with a winning turn total will hold and win. What is the probability that the first player will win if both players play optimally?

   (d) Summarize or visualize the policy, and describe it qualitatively in your own words.

3.6 Advanced Projects

3.6.1 Hog

Hog is a variation of Pig in which players have only one roll per turn, but may roll as many dice as desired. If no 1's are rolled, the sum of the dice is scored. If any 1's are rolled, no points are scored for the turn. It is as if a Pig player must commit to the number of rolls in a turn before the turn begins. (See exercise 2 on the Gambler's Fallacy.)

For a goal score of 100, we recommend a simplifying assumption that a player may roll up to 30 dice. The state transition probabilities can be computed once initially using dynamic programming. (Outcome probabilities for $n+1$ dice can be computed by taking outcome probabilities for n dice, and considering the effects of the 6 possible outcomes of one more die.)

If one graphs the optimal number of dice to roll for each (i, j) pair, one will notice a striking similarity to the shape of the optimal roll/hold boundary for Pig (see the Game of Pig website[12]).

[10] See URL `http://www.hasbro.com/common/instruct/PassThePigs.PDF`.
[11] See URL `http://members.tripod.com/%7Epasspigs/prob.html`.
[12] See URL `http://cs.gettysburg.edu/projects/pig/index.html`.

3.6.2 Ten Thousand

Among dice games, Ten Thousand is what we call a *jeopardy race game*. "Jeopardy" refers to the fact that each turn we are putting our entire turn total at risk. "Race" refers to the fact that the object is to be the first to meet or exceed a goal score. Pig is the simplest jeopardy race game. Most other jeopardy race games are variations of the game Ten Thousand.

In such games, players roll a set of dice (usually 6), setting aside various scoring combinations with each roll that increase the turn total until a player either (1) holds and scores the turn total, or (2) rolls the remaining dice such that there is no possible scoring combination and thus loses the turn total. Generally, if all dice are set aside in scoring combinations, then the turn continues with all dice back in play.

Rules for Ten Thousand can be found in [5] and also online at Dice-Play's Ten Thousand page[13].

Ten Thousand has much in common with Pig, and can also be solved with value iteration. However, writing a program to compute the 2-player solution is more difficult for the following reasons:

- The state space is larger. In addition to keeping track the player score, opponent score, and turn total, one needs to keep track of which subset of 6 dice are in play.
- There are more actions. A player only needs to score one combination per roll, and can therefore possibly choose a subset of several possible scoring combinations.
- There are more possible state transitions. In rolling multiple dice, the many outcomes increase the computation necessary in order to recompute value estimates.
- Greater rule complexity leads to more complex equations.

These factors combine to make this a more advanced programming project. At the time of writing, optimal 2-player play of Ten Thousand is an open research problem. If one wishes to use value iteration to solve a different, simpler jeopardy dice game similar to Pig, additional variations are described in the appendix of [8], available at the Game of Pig website[14].

4 Reinforcement Learning Explorations

Sutton and Barto's *Reinforcement Learning: an introduction* [11] is an excellent starting point for an advanced undergraduate or graduate exploration of reinforcement learning.

The authors classify reinforcement learning methods in three categories, dynamic programming methods, Monte Carlo methods, and temporal-difference learning methods. In their text, "dynamic programming" is defined more generally so as to include value iteration[15]. Dynamic programming and value iteration make a strong assumption that the programmer has a complete model of state transition probabilities and expected rewards for such transitions. One cannot approach a solution of Pass the Pigs (Exercise 8) without a probability model gained through experience.

By contrast, Monte Carlo methods improve state value estimates through simulation or experience, using entire simulated or real games to learn the value of each state. In some cases, a complete probabilistic model of the system is known, yet it is easier to simulate the system than to express and solve equations for optimal behavior.

Temporal-difference (TD) learning methods blend ideas from value iteration and Monte Carlo methods. Like value iteration, TD learning methods use previous estimates in the computation of updates.

[13] See URL `http://homepage.ntlworld.com/dice-play/Games/TenThousand.htm`.

[14] See URL `http://cs.gettysburg.edu/projects/pig/index.html`.

[15] Most algorithm text authors define dynamic programming as a technique for transforming recursive solutions. Recursive solutions require acyclic dependencies.

Like Monte Carlo methods, TD learning methods do not require a model and are informed through experience and/or simulation.

The author has applied methods from each of these categories to the solution of Pig, and has found Pig a valuable tool for understanding their tradeoffs. For example, value iteration is advantageous in that all states are updated equally often. Monte Carlo and TD learning methods, by contrast, update states as they are experienced. Thus, value estimates for states that occur with low probability are very slow to converge to their true values.

If the reader has enjoyed learning the power of these simple techniques, we would encourage continued study in the field of reinforcement learning. With a good text, an intriguing focus problem, and a healthy curiosity, you are likely to reap a rich educational reward.

References

[1] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, USA, 1957.

[2] Dmitri P. Bertsekas. *Dynamic Programming: deterministic and stochastic models*. Prentice-Hall, Upper Saddle River, New Jersey, USA, 1987.

[3] Dan Brutlag. Choice and chance in life: The game of "skunk". *Mathematics Teaching in the Middle School*, 1(1):28–33, April 1994. ISSN 1072-0839.

[4] Ruma Falk and Maayan Tadmor-Troyanski. THINK: a game of choice and chance. *Teaching Statistics*, 21(1):24–27, 1999. ISSN 0141-982X.

[5] Reiner Knizia. *Dice Games Properly Explained*. Elliot Right-Way Books, Brighton Road, Lower Kingswood, Tadworth, Surrey, KT20 6TD U.K., 1999.

[6] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the 11th International Conference on Machine Learning (ICML'94)*, pages 157–163, San Francisco, CA, USA, 1994. Morgan Kaufmann.

[7] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, New York, USA, 1997.

[8] Todd W. Neller and Clifton G. M. Presser. Optimal play of the dice game pig. *UMAP Journal (Journal of Undergraduate Mathematics and Its Applications)*, 25(1), Spring 2004.

[9] Jason A. Osborne. Markov chains for the risk board game revisited. *Mathematics Magazine*, 76(2):129–135, April 2003.
http://www4.stat.ncsu.edu/%7Ejaosborn/research/osborne.mathmag.pdf.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: a modern approach, 2nd ed.* Prentice Hall, Upper Saddle River, NJ, USA, 2003.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an introduction*. MIT Press, Cambridge, Massachusetts, 1998.

[12] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior, 1st edition*. Princeton University Press, Princeton, New Jersey, USA, 1944.

[13] Phil Woodward. Yahtzee®: The solution. *Chance*, 16(1):10–22, 2003.