Pig Analysis

Andrew Pennebaker

December 14, 2010

`https://github.com/mcandre/pig`

Overview

Pig is an old dice game for two or more players. They need not be human, as in an automated monte carlo simulation. Pig's simplicity lends itself as an educational tool for studying statistics, game theory, and numerical analysis.

Game Requirements

- A six-sided die (1d6)
- A scoring mechanism (e.g. pencil and paper)

Objective

Be the first player to score 100 points.

Rules

On a turn, a player continually rolls 1d6, recording the pips as runs, until either

- The player rolls 1 (pigging)
- The player decides to stop rolling

If the player pigs, the turn ends and the next player gets to roll.
Otherwise, the sum of the run is added to the player's score.

Strategies

There are several strategies for playing Pig, each with their advantages and disadvantages.

Always Hold

$points/turn = 0$

$E[turns] = \infty$

This strategy never wins. Fortunately, skipping affords more time to other players.

Always Roll

$p(pigging) = \frac{1}{6}$

$p(scoringRoll) = 1 - p(pigging) = \frac{5}{6}$

$0.5 = 1 - p(scoringRoll)^{E[runLength]}$

$E[runLength] = \frac{ln0.5}{ln\frac{5}{6}} > 3.5$

$E[turns] = \infty$

As with Always Hold, Always Roll never wins, insanely pigging even when a lucky run would total 100+ and win the game. Additionally, Always Roll requires an average of 4 rolls before pigging and passing the turn. Always Roll can halt the game for arbitrarily long turns.

100 or Bust

A minor variation on Always Roll, this strategy attempts to win by rolling a 100+ point run in a single turn.

$E[rollScore] = \frac{0+2+3+4+5+6}{6} = \frac{10}{3} > 3$

$E[winningRunLength] = \frac{100}{E[rollScore]} = 30$

$p(winningRun) = p(scoringRoll)^{30} = (\frac{5}{6})^{30} < 1\%$

$0.5 = 1 - p(winningRun)^{E[turns]}$

$E[turns] = ln_{(\frac{5}{6})^{30}}0.5 < 8$

As unlikely as 100 or Bust is to win on any given turn, the chance of winning in n turns dramatically increases. 100 or Bust is not the best strategy, but it does win games every now and then.

Roll Once

$E[turns] = \frac{100}{E[rollScore]} = 30$

This strategy is simple to implement. Intuitively, it seems like it would win more often than 100 or Bust. However, Roll Once is actually too conservative. $\frac{10}{3}$ points per turn is too few, and 100 or Bust will tend to reach 100 more quickly and therefore win more often.

Roll n

Rolling only once per turn is suboptimal because $p(pigging) = \frac{1}{6}$. Rolling more times per turn will gain more points, balancing against increasing odds of pigging.

$E[scorePerTurn] = 4n \cdot p(scoringRoll)^n$

$E[scorePerTurn; n = 1] = \frac{10}{3} > 3$

$E[scorePerTurn; n = 2] = \frac{50}{9} > 5$

$E[scorePerTurn; n = 3] = \frac{125}{18} > 6$

$E[scorePerTurn; n = 4] = \frac{625}{81} > 7$

$E[scorePerTurn; n = 5] = \frac{15625}{1944} > 8$

$E[scorePerTurn; n = 6] = \frac{15625}{1944} > 8$

$E[scorePerTurn; n = 7] = \frac{546875}{69984} < 8$

$\ldots$

$E[scorePerTurn; n = 26] < 1$

n is optimal at 5 or 6. Since the average score per turn for these is the same, the player might as well choose 5 rolls instead of 6 to save time.

A strategy of rolling 26 times per turn would average less than 1 point per turn because the odds of pigging in 26 rolls is so high, canceling out any run points.

Roll k

This strategy defines k as a value relative to the state of the game. For example, k could be proportional to the current player scores.

$maxPlayerScore = 98$

$p(maxPlayerWinsNextTurn) = p(scoringRoll) = \frac{5}{6}$

In this situation, some players have greater odds of winning–they could win next turn in one roll. In Pig, the only way to win is to win before they win. In other words, to defeat an opponent with a score of 98, another player must win first. Therefore, it is optimal for losing players to roll until they win, employing the 100 or Bust strategy. As long as an opponent player will likely win on their next roll, players have nothing to lose.

k could be relative to the max player score. At the beginning of the game, when players start with score 0, k might be 5, the optimal value from the Roll n strategy. Towards the end, when other players near 100, k increases, approaching ∞, the 100 or Bust strategy.

Conversely, when a player is winning and close to 100 points, k decreases to 1, the Roll Once strategy. At $score = 98$, the next roll will either win the game or pig.

Monte Carlo

An automated simulation for evaluating Pig strategies was implemented in Haskell. As a pure, functional, type-safe language, Haskell may seem unequipped to handle rolling dice, a necessarily nonpure computation (either stateful for pseudorandom number generators or I/O intensive for natural random number generation). In fact, Haskell is up to the task. A custom *roll* function returns a random number from 1 to 6.

```
roll :: IO Int
roll = runRVar (choice [1..6]) DevRandom
```

Some strategies are harder to implement than others, but a Pig strategy is nonetheless a function which takes a subset of the information in the game state and decides the next move. In reality, game state includes various and sundry information, such as whether lunch is nearly over or whether there's a cute girl nearby. Metastrategies such as substituting trick dice or bullying players into skipping turns would significantly add to simulation complexity. For the purposes of the monte carlo, the game information reduces to the player's current scores and the status of the current run. Pig is such a simple game that there are only two moves: roll or hold.

```
data Move = Roll | Hold

type Strategy = [Player] -> [Int] -> Move
```

A player can be modeled with three variables: name, strategy, and current score. With extra effort, these could be treated as two variables, where a strategy encodes its own name programmatically for reference purposes.

```
data Player = Player {
                name :: String,
                strategy :: Strategy,
                score :: Int
        }
```

In detail, the information available to a player consists of a rotating list of players, where the first player in the list is the current player, and a list of the current player's run (an empty list if the player has not yet rolled).

The Always Hold strategy disregards game state and always holds; a constant function. Haskell uses an underscore (_) to denote conditions irrelevant to a computation. Both player list and run list are irrelevant, so they may be underscored from the function inputs. Haskell uses double colons (::) to specify types for robustness, clarity, and code generation. Many of them can be omitted (Haskell has a powerful type inference system), but the signatures shall remain here for readability.

```
alwaysHold :: Strategy
alwaysHold _ _ = Hold
```

Similarly, the Always Roll strategy disregards game state.

```
alwaysRoll :: Strategy
alwaysRoll _ _ = Roll
```

The 100 or Bust strategy is implemented using list pattern matching (:) to extract the current player and guards (|) as syntactic sugar for conditionals. There is a subtle bug: Should a simulation run with no players, the *hundredOrBust* strategy will return an error as it is not defined for a list without at least one players. The simulation can gain robustness by adding a clause after the signature to handle the case where the player list is empty ([]).

```
hundredOrBust :: Strategy
hundredOrBust (p:ps) rs
        | score p + sum rs >= 100 = Hold
        | otherwise = Roll
```

Arbitrary strategies can be implemented once Haskell proficiency is achieved.

Creating players and their strategies is boilerplate coding, but boilerplate can be minimized with the use of default constructors (and further minimized with Lisp macros).

```
defaultPlayer :: Player
defaultPlayer = Player {
                name = "Player",
                strategy = roll5,
                score = 0
        }

rb = defaultPlayer { name = "Roll Bad K", strategy = rollBadK }
```

A default player is assumed to be using the Roll 5 strategy, and all players start the game with 0 points. Here, a player employs a particularly suboptimal strategy, Bad k, the opposite of Roll k. The player will begin by rolling too conservatively, staying behind his opponents' scores, and in the unlikely event his score surpasses his peers', he will roll recklessly when he is winning, handing the game back to them.

Statistical analysis of a simple folk game may be overkill. In any case, the rest of the code is tedium: shuffling players and running $n = 10,000$ simulated games. n appears to be large enough that running $n+$ simulations would produce similar results, $\pm 1\%$.

```
$ make
$ ./pig
Running 10000 games...
Totaling wins...

Roll K Times    30%
Roll Six        30%
Roll Five       27%
100 or Bust     8%
Roll Bad K      3%
Always Hold     0%
Always Roll     0%
Roll Once       0%
```