RESCIENCE C

Replication / Computational Game Theory

[Re] Replication of Pig Optimal Strategy from 'Optimal Play of the Dice Game Pig'

Mark Holcroft[1], Harry Ellingham[1], Billie-Jo Powers[1], and Jasmine Burgess[1]

[1]STOR-i CDT, Lancaster University, Lancaster, United Kingdom

Edited by

ID

Reviewed by

ID

ID

Received

Published

DOI

1 Introduction

1.1 The game of Pig

Pig is a simple dice game of chance, whose interesting probabilistic features have been well-studied in a vast number of academic papers. This report re-examines such a paper by Todd W. Neller and Clifton G.M. Presser [1]. The aim of their paper is to find the optimal policy a player of the game Pig should use, and they calculate this using the value iteration algorithm. At the time of the publishing of the paper, classical value iteration was seen as too slow and state values took too long to converge, and hence a 'layered' approach of working backwards was used. This works by using value iteration on subsets of the state space, ensuring their values converge before moving to the next subset. The paper found that the optimal policy is non-smooth and includes unusual and unintuitive features, which our paper aims to recreate. Throughout the paper, we refer to plots from the original paper as Figures, and our reproductions of these are referred to as Replications.

1.2 The 5R's

In this report, we will be assessing both [1] and our own work with regard to the 5Rs: replicable, re-runnable, repeatable, reproducible, and reusable. A summary of our assessment is as follows:

- **Replicable**: The original publication provides ample pseudo-code and methodology which aids the replicability of their results. Some minor discrepancies are found regarding their score analytics, but overall the majority of results can be successfully replicated. Due to the nature of our work, we have provided similar methodology, and so hope the same level of replicability is present.
- **Re-Runnable**: We are unable to give extensive commentary as to the original publication in this regard, as source code is not available for the majority of the results provided. Code for the game of piglet was made available after communication with authors, which was written in Java. In regard to our work, the code found on the linked repository can be run using Python 3.12.0 (the latest python at time of writing).

Correspondence should be addressed to ()
The authors have declared that no competing interests exist.
Code is available at https://github.com/Harry-Ell/609-CW3 – DOI None. – SWH None.
Data is available at None – DOI None.
Open peer review is available at .

- **Repeatable**: Again, due to the lack of available source code from Neller and Presser, we are unable to comment on how repeatable their results are. However, in our code, the results are all consistent across multiple runs of the program.
- **Reproducible**: Since reproducible is normally referred to in the context of reusing original code, our commentary on the original publication is again limited for this R. However, the piglet code could successfully be converted into python and used to create the same results. All other results could be argued as not reproducible due to the lack of available source code. Our work has been used across all of the authors computers with the same results, which we believe indicates a good level of reproducibility.
- **Reusable**: Reuseable is perhaps one of the harder R's to assess in this context. Our provided code should be reusable in the sense that the code is available via *the github*, for ease of use. In terms of any required modification, the rules of the game of pig can be easily altered in our code, however we cannot comment on how successful one would be trying to implement a variation of Pig such as Pig Dice.

2 Methods

2.1 Value Iteration

The game Pig can be modelled as a Markov Decision Process (MDP), with a state space S comprising all feasible combinations of {Player Score, Opponent Score, Turn Score}, and an action space A = {Roll, Hold}. As in any game of chance, Pig contains a stochastic element, and it is for the player to decide what their policy will be to maximise their probability of winning. Such problems can be solved by an algorithm known as Value Iteration. The value iteration algorithm is shown in 1, with a detailed description of the variables and functions used in the algorithm, and a more in-depth overview of its workings, contained in Appendix A 5.

Algorithm 1 Value Iteration

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

2.2 Layered Value Iteration

Whilst convergence of the value iteration algorithm is guaranteed for $\gamma < 1$ and bounded rewards [2], it can become slow when the state and action spaces are large. For this, the paper offered a solution, whereby the value iteration is implemented in steps, working from the later stages of the game (which sit closer to the reward states of winning), and propagating backwards, ensuring convergence at each step before moving onto the next subset of states. The subsets of states were defined by the sum of player and opponent scores - the first subset being {99,99}, corresponding to a sum of 198, and the next subset being { {99, 98}, {98, 99} }, for a sum of 197, and so forth until the sum reaches zero. This

is much quicker than the standard value iteration algorithm, as the values for later states are being updated using fixed constants of values which have already converged.

2.3 Re-Implementation

In our reproduction, we used the layered value-iteration algorithm which was used in the original paper. This was created in Python 3.12.0, and the package numba was used to reduce runtime by recompiling loops in C [3]. The figures and metrics were reproduced using several methods: Figure 3 was drawn directly from the optimal policy found by the value iteration algorithm, whereas Figures 4, 5, 6, 7 and the metrics were found using simulation, again using the optimal policy. Each of the simulations used 10^6 realisations and seed 123. The plots were created using various features included in the Plotly package in Python [4].

3 Results

3.1 Piglet Convergence Results

Our first reproduction efforts were made in reproducing Figure 2 from [1], which depicts the convergence rates for a game of Piglet up to a score of 2. Note that Piglet is a simplified version of Pig, in which a coin is used in place of a die. This reproduction is shown in Replication 1. Here, we have managed to achieve the same convergence rates as the original paper, and have critically been able to converge to the same final state values.

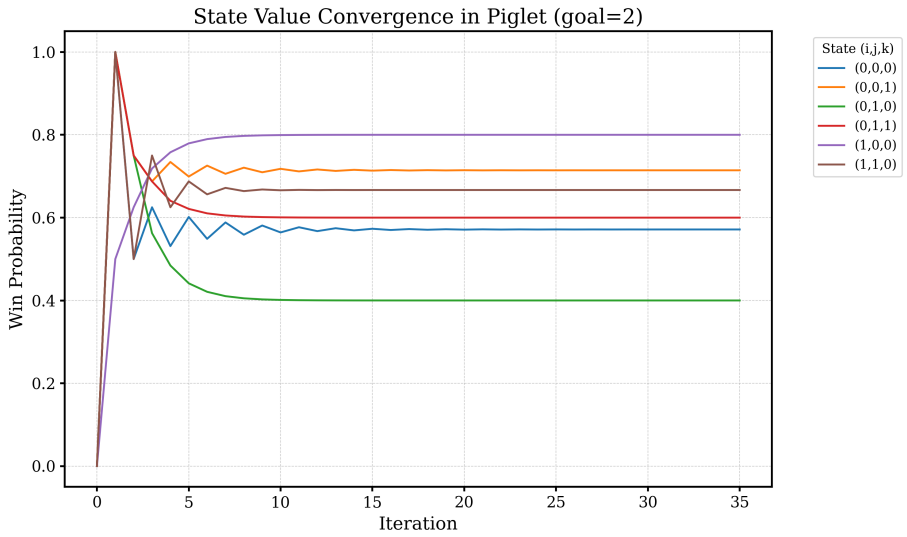
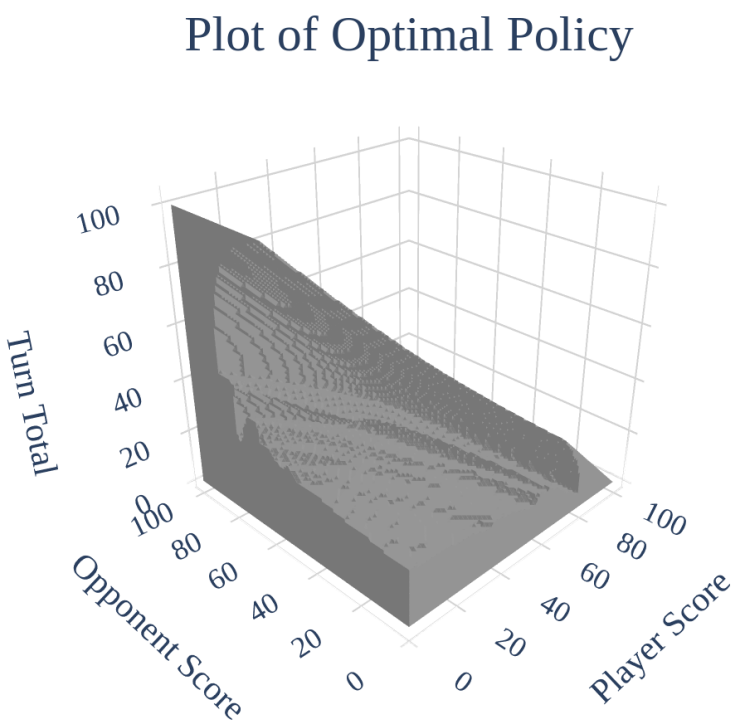


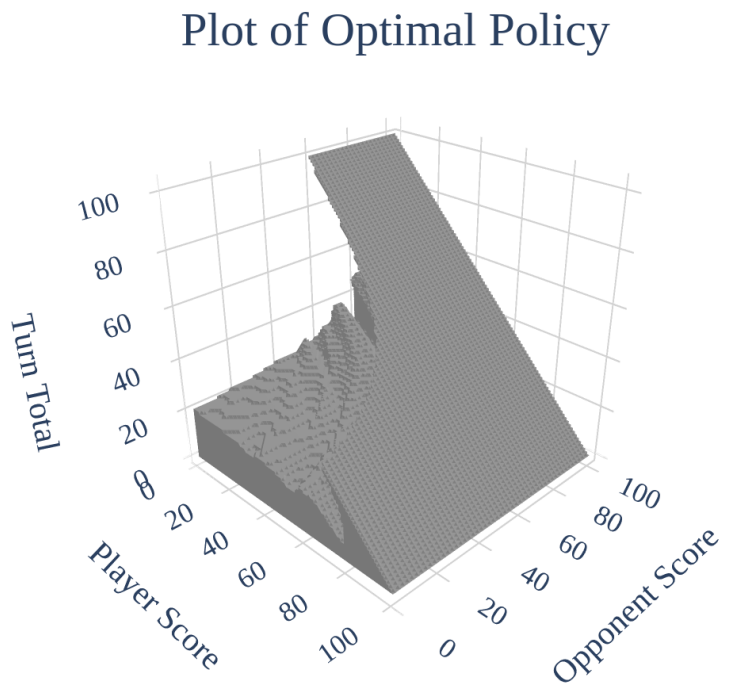
Figure 1. The convergence of state values for the game Piglet with a target score of 2, reproduced from [1].

3.2 Graphing the Optimal Policy

A core result from the original paper which we were able to plot is shown in Figure 2. This has distinct similarities with the surface shown in the original paper, and indicates a certain level of reproducibility.



(a) The optimal policy from angle 1.



(b) The optimal policy from angle 2.

Figure 2. The optimal policy for any combination of player score, opponent score, and turn score in a game of Pig. The surfaces show the points at which we stop rolling and choose to hold.

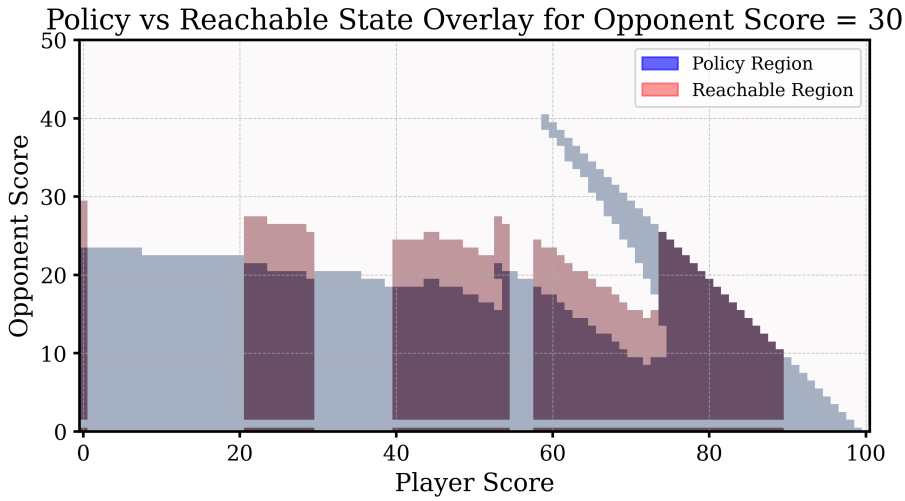


Figure 3. The reachable states and optimal policy for a player given that the opponent is on a score of 30, as compared to the heuristic "hold at 20".

3.3 Replication of Figure 4

Another graph which was successfully replicated was Figure 4. Shown in Figure ??, it depicts a cross-section of the reachable states for an opponent score of 30, and includes the optimal boundary and the "hold at 20" heuristic as a benchmark for comparison. The graph was produced using simulation, and shares the same features and properties with that of the original paper. Note crucially that the reachable states are states which are reachable during any stage of the game - not only when the opponent is on a score of 30.

3.4 Replication of Figures 5 & 6

Figures 5 and 6 of the original paper depict the optimal policy at all reachable states. Note here that the reachable states are calculated based on the assumption that the opponent could be playing using any strategy - if they were to be playing optimally, these figures would be symmetric about the axis Player Score = Opponent Score. These figures have been reproduced in Figure 4. They show clear similarities with those of the original paper, particularly the omission of states involving player scores of between 1 and 19, and the tendency to increase the maximum turn score for larger values of opponent score.

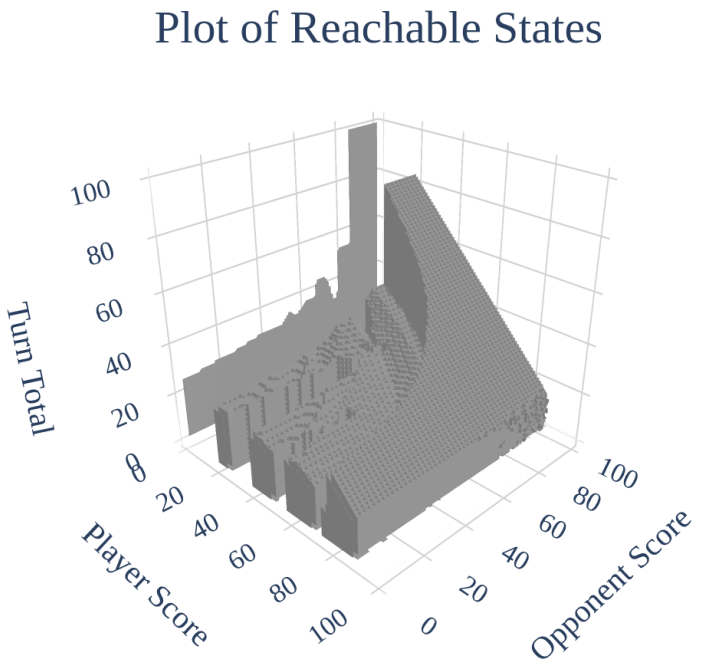
3.5 Replication of Figure 7

The final figure which we reproduced was Figure 7. This resembles very closely what was found in the original paper, with the contour featuring similar gradients across the graph.

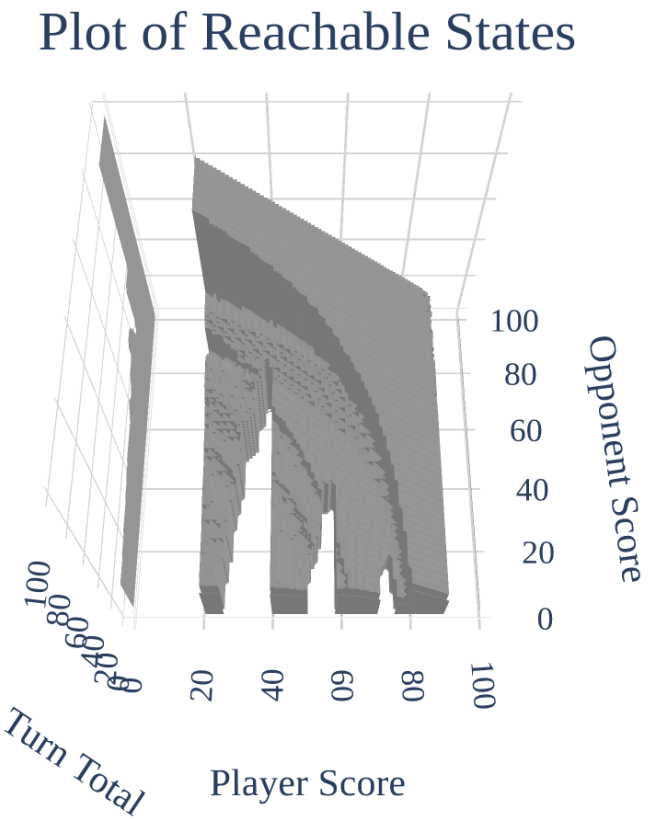
3.6 Key Metrics

In addition to the plots, we also replicated some of the key metrics given by the paper. This offered an additional insight regarding the accuracy of our reproduction. The metrics were found through simulation, using 10^6 samples and the seed 123. The metrics we replicated, and the values achieved, are as follows:

- Piglet state values, with a target score of 2.



(a) 3D view of states reachable by an optimal Pig player.



(b) The replication of Figure 5. The replication was done through simulation, and captures similar patterns to those found in the original paper.

Figure 4. Two views of states reachable by an optimal Pig player.

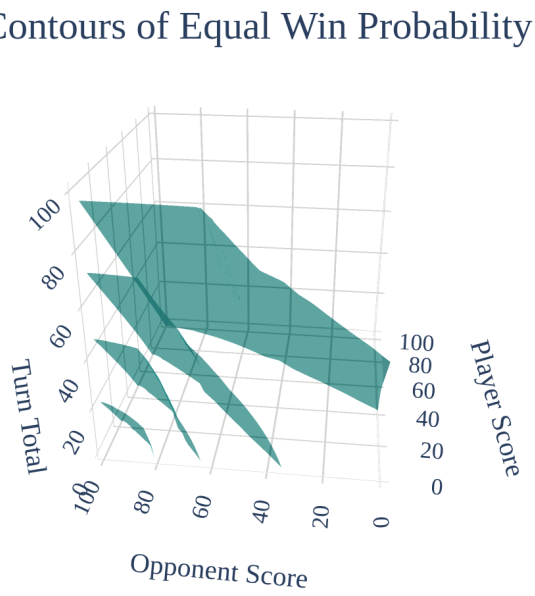


Figure 5. The win probability contours for probabilities (3%, 9%, 27% and 81%. These mirror very accurately the contours given by the original paper.

  - The final values of our states were found to be:
    * $P_{0,0,0} = \frac{4}{7}$
    * $P_{0,0,1} = \frac{5}{7}$
    * $P_{0,1,0} = \frac{2}{5}$
    * $P_{0,1,1} = \frac{3}{5}$
    * $P_{1,0,0} = \frac{4}{5}$
    * $P_{1,1,0} = \frac{2}{3}$

    These aligned exactly with those given in the original paper.

- The win probability of the optimal strategy against an opponent using the "hold at 20" heuristic.
  - The win probability when the optimal policy plays first was found to be 0.5721, with a 95% confidence interval of $[0.5696, 0.575]$. This compares with the value of 0.5874 given by the paper. Clearly, our result is significantly lower than that given by Neller and Presser. However, we did find that when employing a "hold at 22" policy, with the optimal player going first, they win 0.58565 of the games, with a 95% confidence interval of $[0.583, 0.589]$, which coincides with the results of Neller and Presser. Since the method used to obtain their results is not specified, it is difficult to comment on the reasons for these discrepancies, but these findings suggest a lack of replicability for this particular section of the original paper.
  - The win probability when the optimal policy plays second was found to be 0.49518, with a 95% confidence interval of $[0.492, 0.498]$. By comparison, the original paper stated a value of 0.4776. Again, we note that our results differ significantly from Neller and Presser's; with the "hold at 22" policy giving a closer value of 0.47895, with a 95% confidence interval of $[0.476, 0.482]$.
- The win probability of the optimal strategy against an opponent playing optimally.

- The win probability when the optimal policy plays first was found to be 0.53047%, with a 95% confidence interval of $[0.527, 0.534]$. This compares well with Neller and Pressers value of 0.5306, indicating that this result is replicable.

Note that there were distinct similarities between our findings and those of the original author, although there remained some discrepancies of varying degrees. We conclude a likely reason for this is that the values in the original paper may not have fully converged - this is particularly due to the use of older systems and potentially the use entirely of a slower system such as Python (without the numba package to allow recompiling in C).

4 Conclusion

Our aims when beginning this report were two-fold - to assess whether the original paper was correct in its findings, and to determine how easily and to what degree these results could be reproduced. Based on our graphical output and simulation metrics, it is evident that the results are of a high quality and, accounting for a margin of error that could be attributed to a lack of convergence, correct. This is especially true of the optimal policy, from which all other results can be derived.
Regarding reproducibility, there are certain areas in which the report could have eased the reproduction process. The first of these is a lack of clarity in how the plots were made - whilst it was possible to replicate all figures using Python, this process involved fine-tuning methods designed for other purposes.. For example, Figure 4 was recreated using the Isosurface feature within Plotly, whose primary purpose is plotting 3D surfaces, not volumes. It was also sometimes unclear precisely what had been plotted - this was particularly evident in Figure 4, where it was not immediately clear how the reachable states were defined.
Other than the specific plots and metrics, the inclusion of such details as the software and any packages used were specified, including their version. Another improvement would be to indicate what seed was used and how many realisations were made such that simulation results could be compared exactly.
Despite some minor challenges, the replication was ultimately successful, with all key graphs and primary metrics accurately reproduced. In this regard, the original paper's findings were validated, particularly concerning the optimal policy from which all other results derive. However, to enhance reproducibility, future works should place greater emphasis on clearly specifying the methods used to produce figures, detailing the software environments and package versions, and explicitly stating simulation settings such as random seeds and the number of realisations. These small but impactful improvements would significantly support transparent and exact replication efforts.

5 Appendix A

Definitions of Pseudocode Components

Sets and Spaces

- $\mathcal{S}$: Set of all possible states.
- $s \in \mathcal{S}$: A specific state from the state space.
- s': A successor (next) state.

Functions and Variables

- $V(s)$: Current value estimate of state s.
- v: Temporary variable to store the old value of $V(s)$.
- Δ: Tracks the maximum change in the value function across states in an iteration (used to check for convergence).
- ϵ: Convergence threshold; iteration stops when $\Delta < \epsilon$.
- γ: Discount factor ($0 \leq \gamma \leq 1$), determines the importance of future rewards.

Transition and Reward Functions

- $\mathcal{P}^a_{ss'}$: Probability of transitioning from state s to state s' when action a is taken.
- $\mathcal{R}^a_{ss'}$: Expected immediate reward received after transitioning from state s to s' using action a.

Operators

- $\max_a$: Selects the action a that maximizes the expression.
- $\sum_{s'}$: Sum over all possible next states s'.
- $\leftarrow$: Assignment operator (updates the value of a variable).
- $\max(x, y)$: Returns the maximum of values x and y.
- $|v - V(s)|$: Absolute difference between the old and new value of state s (used to compute convergence).

Algorithm Steps

1. **Initialization**: Assign an arbitrary value to each $V(s)$ (can be zero or random values).
2. **Iteration (Repeat-Until Loop)**:
   - Start each iteration with $\Delta \leftarrow 0$ (reset maximum value change).
   - For each state s:
     - Store the current value in v.
     - Update $V(s)$ using the Bellman optimality equation.
     - Update Δ with the maximum observed change.
   - Repeat until $\Delta < \epsilon$ (i.e., the values have converged).

6 References

References

1. T. W. Neller and C. G. Presser. "Optimal Play of the Dice Game Pig." In: **The UMAP Journal** 25.1 (2004), pp. 25–47.
2. R. S. Sutton and A. G. Barto. **Reinforcement learning : an introduction**. eng. Cambridge, Massachusetts, 2018.
3. S. K. Lam, A. Pitrou, and S. Seibert. "Numba: A LLVM-based Python JIT Compiler." In: **Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC** (2015), pp. 1–6. DOI: 10.1145/2833157.2833162.
4. P. T. Inc. **Plotly: The interactive graphing library for Python**. Version accessed: May 2025. 2015. URL: https://plotly.com/python/.