PANTHÉON-ASSAS
UNIVERSITÉ
PARIS

Arsen Pidburachynskyi

Automated Simulation and Analysis of Non-trivial Bayesian Dialogues

Supervisor: Christina Pawlowitsch

2023

Abstract

This paper introduces an algorithm for the automatic simulation of Bayesian dialogues, implemented in Python. The algorithm allows for the generation of dialogues that embody Bayesian reasoning. In addition to detailing the algorithm's implementation and key features, we explore the probability of the emergence of non-trivial Bayesian dialogues by conducting random simulations on a vast sample of dialogues. The evaluation of the algorithm provides insights into its performance and the effectiveness of its simulations, underscoring its capability to encapsulate the core of Bayesian reasoning.

1 Introduction

I begin by expressing my profound gratitude to Christina Pawlowitsch and Daniel Toneian. Their astute discussions and insights have been instrumental in the evolution of my research. Notably, their guidance on the generation of random partitions, ensuring equal probabilities for each partition, has been foundational. Their expertise has provided pivotal directions during the formative stages of this study, and for that, I am deeply indebted.

Bayesian dialogues offer a intersection of probability and conversational dynamics, illustrating how beliefs are adjusted and refined in response to new information. While this can be studied theoretically, translating these dialogues into a large-scale simulation offers unique insights that are otherwise challenging to discern.

Central to this exploration is the development of a Python algorithm, the primary tool in this research. While the concept of generating dialogues might seem deceptively simple at a glance, the underlying algorithmic design is far from trivial. The algorithm doesn't merely churn out dialogues; the system generates dialogues with complete randomness, ensuring an equal probability for each unique dialogue. Remarkably, even within this randomness, the generated dialogues manage to capture the complexities and nuances of iterative Bayesian communication.

This research draws motivation from the work of Geanakoplos, J. and Polemarchakis, H.(1982). Their insights into Bayesian convergence provide the theoretical underpinning that the algorithm seeks to emulate and expand upon. By integrating their concepts, the algorithm aims to replicate, on a broad scale, the cyclical communication patterns intrinsic to Bayesian dialogues.

Once a substantial collection of dialogues is generated, the core analysis begins. The research's primary analytical objective is to discern the occurrence frequency of non-trivial dialogues within this vast sample. Such dialogues are those that diverge from predictable patterns, adding layers of complexity or unexpected twists.

Subsequent sections of this paper delve deeper into the algorithm's inner workings. The Python code's intricacies are elaborated upon, providing readers with a clear picture of its architecture, decision-making processes, and the foundational logic it rests upon. Additionally, the paper elucidates the simulation's methodology, shedding light on how dialogues are classified as 'non-trivial' and the metrics employed for the analysis.

In essence, this research, while primarily presenting a data-driven analysis of non-trivial dialogues in Bayesian simulations, also introduces and explains a Python algorithm tailored for this exact purpose.

2 Theoretical presentation of Bayesian Dialogues

Bayesian dialogues involve iterative communications and revisions between agents, leading to convergence of their beliefs to a shared posterior distribution. Geanakoplos, J. and Polemarchakis, H.(1982), provides insights into the dynamics of Bayesian dialogues and the conditions under which convergence occurs.

Bayesian reasoning is a framework for updating beliefs based on prior knowledge and new evidence. It involves calculating posterior probabilities using Bayes' theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where $P(A|B)$ is the posterior probability of event A given evidence B, $P(B|A)$ is the likelihood of evidence B given event A, $P(A)$ is the prior probability of event A, and $P(B)$ is the probability of evidence B.

In Bayesian dialogues, agents start with different initial information and beliefs. However, they share common priors, which represent their collective knowledge before any evidence is considered. Additionally, their information is partitioned into finite sets, representing their individual knowledge states.

Agents engage in iterative communications and revisions, updating their beliefs based on new evidence and the beliefs of other agents. They share information through announcements and revise their beliefs using Bayes' theorem. The iterative process continues until an "absorbing state" is reached, where further announcements no longer impact the reduction of possible states.

Upon reaching the absorbing state, the agents' beliefs converge, and their posteriors become identical. This convergence is not arbitrary but follows from the agents' iterative communications and revisions, driven by Bayesian reasoning. Aumann's theorem guarantees that agents' posteriors will be the same at the absorbing state.

Based on the insights from the paper, we have developed a Python algorithm to simulate Bayesian dialogues. The algorithm incorporates the principles of Bayesian reasoning, common priors, finite information partitions, and iterative communications and revisions.

Bayesian dialogues provide a framework for modeling conversations involving probabilistic reasoning and belief updates. Geanakoplos, J. and Polemarchakis, H. (1982) sheds light on the dynamics of Bayesian dialogues, highlighting the conditions under which convergence to a shared posterior occurs.

The simulation of Bayesian dialogues using the Python algorithm provides a practical tool for studying and experimenting with these concepts.

Let's look at mathimatical formalisation of the exchange between the agents,

In Geanakoplos, J. and Polemarchakis, H.(1982), a mathematical process is defined to represent the progression of a dialogue between two agents, denoted as α and β. The authors demonstrate that for a universe of finite size, these agents will inevitably reach an absorbing state, a point in their dialogue where no new information can influence their posterior beliefs.

The dialogue progression is formulated as follows:

Step 1:

1. Agent α announces a value $\mathbf{q}_1^\alpha(\omega)$ which represents their belief about the probability of an event ω given their partition of ω, denoted as $\mathscr{P}^\alpha(\omega)$. We then define a_1 as the set of all partitions for which α's belief matches the actual probability.

2. Agent β observes a_1 and announces their own belief $\mathbf{q}_1^\beta(\omega)$, representing the probability of event ω given their own partition of ω and the partitions in a_1. We define b_1 as the set of all partitions for which β's belief matches the actual probability.

Step t:

1. Agent α announces $\mathbf{q}_t^\alpha(\omega)$, their updated belief about the probability of event ω given their partition of ω and the partitions in b_{t-1}. We define a_t as the set of all partitions for which α's updated belief matches the actual probability.

2. Agent β observes a_t and announces their own updated belief $\mathbf{q}_t^\beta(\omega)$, representing the probability of event ω given their partition of ω and the partitions in a_t. We define b_t as the set of all partitions for which β's updated belief matches the actual probability.

This process continues iteratively. Each agent adjusts their beliefs based on the partitions and probabilities announced by the other. The agents' dialogues form a sequence that converges to an absorbing state, where their beliefs no longer change, as there is no new information from the other agent's posterior beliefs.

The progression of this dialogue can be mathematically formalised as follows.

Step 1:

$$\text{Agent } \alpha \text{ announces } \mathbf{q}_1^\alpha(\omega) = \frac{p\left(\mathscr{P}^\alpha(\omega) \cap A\right)}{p\left(\mathscr{P}^\alpha(\omega)\right)}.$$

$$\text{Define } a_1 = \left\{ k \mid \frac{p\left(P_k^\alpha \cap A\right)}{p\left(P_k^\alpha\right)} = \mathbf{q}_1^\alpha(\omega) \right\}.$$

$$\text{Agent } \beta \text{ announces } \mathbf{q}_1^\beta(\omega) = \frac{p\left(\left(\mathscr{P}^\beta(\omega) \cap \bigcup_{k\in a_1} P_k^\alpha\right) \cap A\right)}{p\left(\mathscr{P}^\beta(\omega) \cap \bigcup_{k\in a_1} P_k^\alpha\right)}.$$

$$\text{Define } b_1 = \left\{ l \mid \frac{p\left(\left(P_l^\beta \cap \bigcup_{k\in a_1} P_k^\alpha\right) \cap A\right)}{p\left(P_l^\beta \cap \bigcup_{k\in a_1} P_k^\alpha\right)} = \mathbf{q}_1^\beta(\omega) \right\}.$$

Step t:

$$\text{Agent } \alpha \text{ announces } \mathbf{q}_t^\alpha(\omega) = \frac{p\left(\left(\mathscr{P}^\alpha(\omega) \cap \bigcup_{l\in b_{t-1}} P_l^\beta\right) \cap A\right)}{p\left(\mathscr{P}^\alpha(\omega) \cap \bigcup_{l\in b_{t-1}} P_l^\beta\right)}.$$

$$\text{Define } a_t = \left\{ k \mid \frac{p\left(\left(P_k^\alpha \cap \bigcup_{l\in b_{t-1}} P_l^\beta\right) \cap A\right)}{p\left(P_k^\alpha \cap \bigcup_{l\in b_{t-1}} P_l^\beta\right)} = \mathbf{q}_t^\alpha(\omega) \right\}.$$

$$\text{Agent } \beta \text{ announces } \mathbf{q}_t^\beta(\omega) = \frac{p\left(\left(\mathscr{P}^\beta(\omega) \cap \bigcup_{k\in a_t} P_k^\alpha\right) \cap A\right)}{p\left(\mathscr{P}^\beta(\omega) \cap \bigcup_{k\in a_t} P_k^\alpha\right)}.$$

$$\text{Define } b_t = \left\{ l \mid \frac{p\left(\left(P_l^\beta \cap \bigcup_{k\in a_t} P_k^\alpha\right) \cap A\right)}{p\left(P_l^\beta \cap \bigcup_{k\in a_t} P_k^\alpha\right)} = \mathbf{q}_t^\beta(\omega) \right\}.$$

Consider the following example:

$$\begin{aligned}
\Omega &= \{1,2,3,4,5,6,7,8,9\}, \quad p(k) = \frac{1}{9} \text{ for all } k \in \Omega;\\
\mathscr{P}^\alpha &= \{\{1,2,3\},\{4,5,6\},\{7,8,9\}\};\\
\mathscr{P}^\beta &= \{\{1,2,3,4\},\{5,6,7,8\},\{9\}\};\\
\mathscr{P}^\alpha \vee \mathscr{P}^\beta &= \{\{1,2,3\},\{4\},\{5,6\},\{7,8\},\{9\}\};\\
\mathscr{P}^\alpha \wedge \mathscr{P}^\beta &= \{1,2,3,4,5,6,7,8,9\};\\
\mathscr{A} &= \{1,5,9\};\\
\omega &= 1;\ \text{hence } \mathscr{P}^\alpha(\omega) = \{1,2,3\} \text{ and } \mathscr{P}^\beta(\omega) = \{1,2,3,4\}.
\end{aligned}$$

Applying the steps defined earlier, we have the following results. Agent α will announce a probability of 1/3, then β will announce a probability of 1/4. Then α will announce 1/3 and β will announce 1/4. Finally, in the third round, they will both announce 1/3, since they've reached a situation where they do not gain any information from each other's posterior beliefs. According to Aumann's agreement theorem, their posterior beliefs should then be equal.

3 Python Implementation

After an in-depth exploration and analysis in our study, I've encapsulated our findings and methodologies into a custom Python script. This is tailored specifically for deducing the posterior probability of a particular event within the Bayesian framework. The heart of this algorithm, inspired by Bayes' theorem, finds its representation in a Python class I've termed `posterior_probability_computation`.

About the `posterior_probability_computation` Class:

- **Purpose**: To compute the posterior probability within the Bayesian reasoning paradigm.
- **Features**:
  - **Flexibility**: Adapted to accept a plethora of parameters, making it apt for varied scenarios.
  - **Iterative Mechanism**: Designed with an iterative approach, continually recalculating probabilities until the beliefs of agents α and β find common ground, indicating an equilibrium.
- **Parameters**:
  - N : Size of the considered universe.
  - α & β : Partitions representing the respective knowledge states of two distinct players.
  - A : The target event for which we are computing the posterior probability.
  - w : An acknowledged event that adjusts the posterior probability of event A.

The `main_run` Function:

- Alongside the class, I've integrated a function termed `main_run` which orchestrates the computation sequence.
- **Roles**:
  - **Initialization**: Engages the `posterior_probability_computation` class.
  - **Configurability**: Taps into a YAML configuration file to retrieve parameters, emphasizing modularity and straightforward application.
  - **Execution**: Sets the computation wheels in motion post instantiation.
  - **Visualization**: Enables users to visually track the ebb and flow of computational steps across iterations.

For those intrigued to delve beneath the surface, the full Python script crafted by me elucidating these functionalities awaits your perusal:

```
import numpy as np

import matplotlib.pyplot as plt

import argparse

import yaml
```

```
class posterior_probability_computation :
    # For the full code of class look at appendix A


if __name__=='__main__':
    with open('config.yaml', 'r') as yaml_file:
        data = yaml.safe_load(yaml_file)
    main_run(data['N'], data['partition_1'], data['partition_2'], data['A'],data['w'], visualisat
```

To validate the Python script, we performed several tests on randomly generated partitions of different sizes. The partitions were generated using an urn model, which is a well-known method for generating random partitions. The implementation of this method in Python is provided below. This approach was adapted from Stam, A.J. (1983), who provides a detailed proof of its validity.

The process begins by defining a discrete distribution based on the Bell numbers, which represent the number of possible partitions for a given set size. This distribution ensures that all partitions have the same probability of being generated.

Next, we sample the number of urns from the defined discrete distribution. The number of urns determines the number of subsets or blocks in the partition.

The code generates a random partition by iterating through the elements to be partitioned and assigning each element randomly to one of the urns. The resulting partition is represented as a list of lists, where each inner list corresponds to an urn and contains the elements assigned to that urn. This list representation allows for convenient access and retrieval of the partitioned elements based on their respective urns.

It's important to note that the detailed proof of the correctness and properties of this partition generation method can be found in the referenced paper by Stam, A.J., 1983. We adapted the method to suit our specific needs and performed empirical tests to validate its effectiveness.

Please refer to the referenced paper for a thorough understanding of the underlying principles and theoretical properties of this partition generation method.

However, it's crucial to mention a notable limitation in the context of our work. The method isn't impeccably suited for generating partitions of sizes greater than a hundred. This limitation stems primarily from certain constraints inherent to the Python programming language. For instance, Python has a maximum size for integers, which can become a bottleneck when handling very large partitions. While this might present challenges for extremely large sets, in the grand scheme of our study, the impact of this limitation is not particularly significant for most practical applications.

```
import numpy as np
```

```
def T(n, N=15):
    # Bell number (N=25 ?)
    total_sum = 0
    for k in range(1, N):
        total_sum += k ** n / np.math.factorial(k)
    return total_sum / np.exp(1)


def define_discrete_dist(u, n):
    # Discrete distribution
    return np.exp(-1) * (u ** n) / np.math.factorial(u)


def sample_discrete_dist(n, u_max=10):
    # Sampling of the number of urns
    density = [define_discrete_dist(u, n) for u in range(u_max)]
    return np.random.choice(a=np.arange(u_max), p=[e / sum(density) for e in density])


def sample_random_partition(n, u_max):
    u = sample_discrete_dist(n, u_max)
    partition = [[] for _ in range(u)]
    for b in range(1, n + 1):
        urn_chosen = np.random.randint(1, u + 1)
        partition[urn_chosen - 1].append(b)
    return partition
```

Following the partitions generation, we carried out two critical tests:

1. We investigated whether any partitions of any size ended in an absorbing state.
2. We checked if the results remained consistent when we permuted the elements of partitions.

Both tests were successful, further validating the effectiveness of our algoritm.

Now let's delve deeper into the results generated by our algorithm. We shall use the same example as presented in the previous section to maintain continuity and foster better understanding. This example consists of the following components:

$$
\begin{aligned}
\Omega &= \{1,2,3,4,5,6,7,8,9\}, \quad p(k) = \frac{1}{9} \text{ for all } k \in \Omega;\\
\mathscr{P}^\alpha &= \{\{1,2,3\},\{4,5,6\},\{7,8,9\}\};\\
\mathscr{P}^\beta &= \{\{1,2,3,4\},\{5,6,7,8\},\{9\}\};\\
\mathscr{P}^\alpha \vee \mathscr{P}^\beta &= \{\{1,2,3\},\{4\},\{5,6\},\{7,8\},\{9\}\};\\
\mathscr{P}^\alpha \wedge \mathscr{P}^\beta &= \{1,2,3,4,5,6,7,8,9\};\\
\mathscr{A} &= \{1,5,9\};\\
\omega &= 1;\ \text{hence } \mathscr{P}^\alpha(\omega) = \{1,2,3\} \text{ and } \mathscr{P}^\beta(\omega) = \{1,2,3,4\}.
\end{aligned}
$$

Let us now examine the output of our algorithm:

The algorithm generates two sequences of probabilities, one for each individual α and β. Each element in these sequences represents the probability associated with the individuals at each round of exchange.

For individual α : [0.3333333333333333, 0.3333333333333333, 0.3333333333333333, 0.3333333333333333]
And for individual β : [0.25, 0.25, 0.25, 0.3333333333333333]

In these sequences, the first element signifies the prior of each individual, the second element represents the probability after the first round of exchange, the third element corresponds to the probability after the second round, and so on.

One of the key features of our algorithm is that it provides a graphical representation of these exchanges. This visualization aids in intuitively understanding the progression and dynamics of the dialogue. The graphical representation for the present case is as follows:

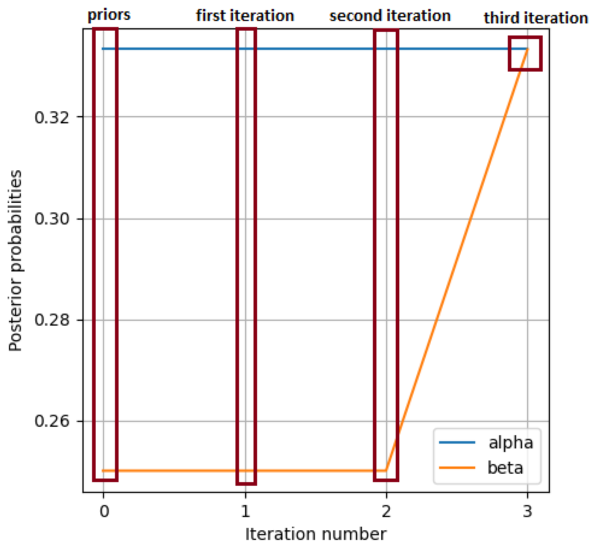

Figure 1: Graphical representation of the dialogue exchange.

The Python script we have developed allows for effective and intuitive computation of the posterior probabilities within a Bayesian framework. The script, built with extensibility and robustness in mind, handles a variety of scenarios with diverse parameters, helping us in the investigation of the complex interplay between the beliefs of two players, α and β, within a given universe.

Our implementation ensures a absorbing state is reached during iterations, where the partitions of the universe for both α and β remain unchanged, implying that the beliefs of the players have been updated to their fullest extent given the available information.

Through several successful tests on randomly generated partitions of varying sizes, and further verification by investigating the consistency of results under permutations of partition elements, the robustness and reliability of the algorithm are ensured.

Moreover, the algorithm is designed to provide visual aid in understanding the progression of the computation, thereby making it a comprehensive tool for such complex probability computations. In summary, our implementation of this Bayesian framework serves as a powerful and reliable tool for the computation of posterior probabilities in the face of uncertain information.

4 Frequency of non-trivial bayesian dialogues

In our examination of Bayesian dialogues, a certain category of discussions, termed as "non-trivial," warranted special attention. Unlike the routine or predictable exchanges, these dialogues presented unique and distinct patterns, often deviating from the anticipated trajectory of a conversation.

Three notable characteristics were identified in our analysis of non-trivial dialogues:

1. **Change of Direction**: An intriguing observation was the instances where the trajectory of an individual's probability evolution demonstrated a pronounced shift.

2. **Convergence to joint partition's elements**: Convergence does consistently occur to elements within the joint partitions. However, the critical distinction is not merely whether individuals' final partitions converge, but to how many shared elements they converge. While convergence to a single element suggests a high degree of agreement in reasoning behind the probability, convergence to multiple elements indicates that, despite a superficial agreement in outcomes, the participants' paths or reasonings to reach those outcomes differ significantly.

3. **Time to Convergence**: The duration or number of iterations taken for beliefs to align provides a measure of the dialogue's progression. It's a metric to understand the time frame within which individuals either realign their beliefs or perhaps find a shared reference point within the conversation.

4.1 Change of directions

Delving deeper into the first characteristic - the Change of Direction - we conducted an extensive set of simulations to quantify the frequency of these shifts. Each set consisted of 200,000 simulations for various N values ranging between 10 and 154, where N represents the set of possible states of the world.

Our observations are summarized as follows:

- **Single Change of Direction**: This pertains to dialogues where we observed only one change in the trajectory of probability evolution for one or both participants. The frequency of such shifts exhibited a pronounced dependence on the size of N. For instance, for N = 10, we observed single change of direction only in 0.70 % of the cases, scaling up or down as the size of N increased.
- **Two Changes of Direction**: Although less frequent, there were instances where the probability evolution underwent two distinct trajectory shifts within a dialogue. However, these occurrences were sporadic, with some N sizes, like N = 22 to N = 26, showing no such dialogues in our 200,000 simulations for each N.

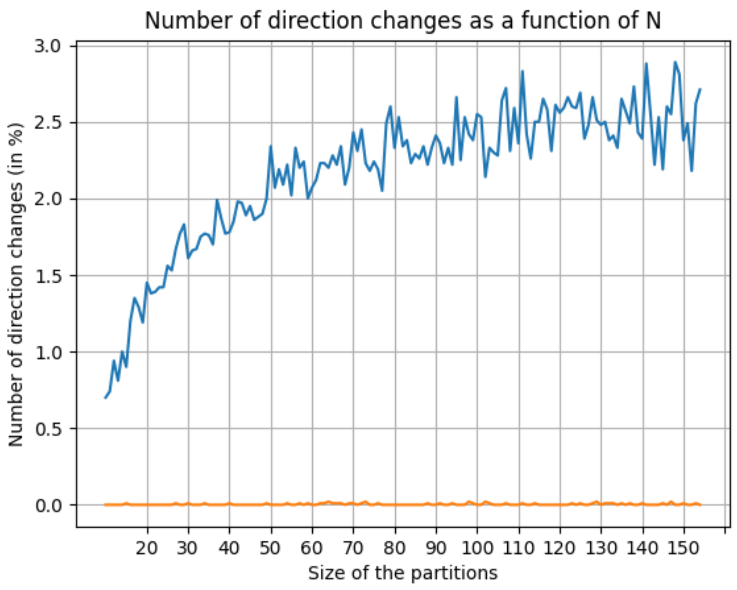

Figure 2: Frequency of Direction Changes in Bayesian Dialogues as a Function of N

This graph effectively visualizes the frequency distribution of these shifts across varying N sizes, providing a comprehensive overview of our findings.

By discerning these patterns, especially the changes in the direction of probability evolution, we gain a deeper understanding of the dynamism inherent in Bayesian dialogues, further emphasizing their non-trivial nature.

4.2 Convergence to joint partition

In our further analysis focusing was on understanding the convergence dynamics concerning shared elements within joint partitions. The investigation revolved around assessing the nature of convergence of individual final partitions with the fluctuating partition size, denoted as N.

The amassed data yielded an intriguing pattern: with an escalation in the partition size, there was a discernible inclination in the mean count of shared elements within joint partitions. This trend, captured and visualized in the following graph, elucidates the intricate dynamics of convergence based on the magnitude of the partition.

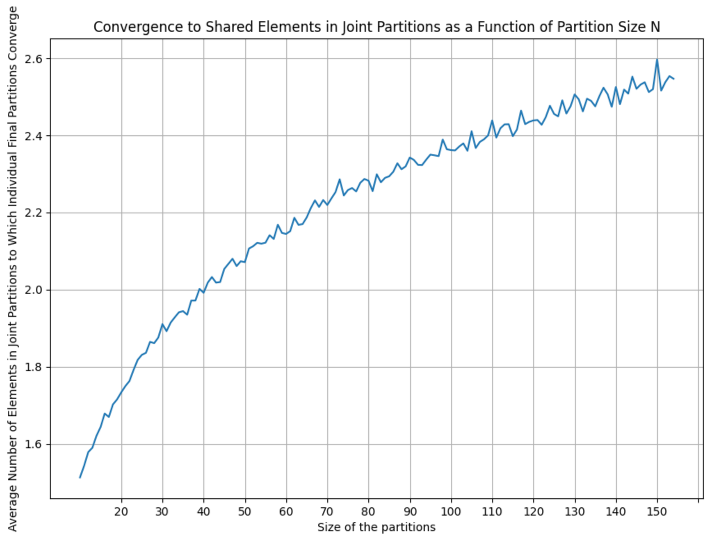

Figure 3: Convergence Trend of Individual Final Partitions as Influenced by Partition Size N

The above illustration delineates the intricate relationship between partition size N and the typical number of shared elements within joint partitions to which individual final partitions gravitate. The data reveals a palpable trend, punctuated occasionally by minor deviations. This suggests an intricate web of underlying dynamics governing the convergence behavior rooted in partition size. The pattern intimates a mounting challenge for individuals in comprehending each other as the information pool swells.

4.3 Iterations in Bayesian Dialogues

In our exploration of Bayesian dialogues, the number of iterations required for convergence emerged as a crucial metric. Through our analysis, we found that on average, Bayesian dialogues necessitated around 2 iterations for beliefs to align or find a shared reference. Significantly, this average was consistent and remained largely unaffected by the size of the partitions. The average number of iterations did not surpass 2.14 or fall below 2, regardless of the partition size.

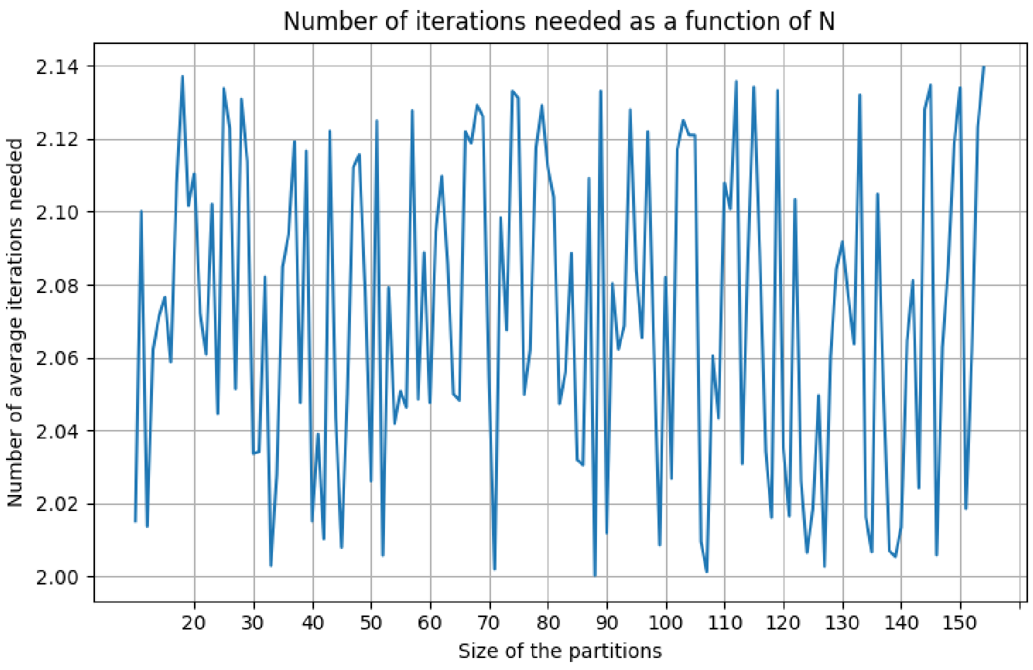

Figure 4: Average number of iterations required for convergence in Bayesian dialogues as a function of partition size.

This stability in iteration count implies that the length of a Bayesian dialogue, in terms of the number of back-and-forths, is not significantly influenced by the size of the underlying partitions.

5 Conclusions

In our comprehensive analysis of Bayesian dialogues, we particularly focused on those categorized as non-trivial. The findings from our systematic study and algorithmic simulations identified three key characteristics inherent to these dialogues:

1. The propensity for unpredictable changes in the direction of an individual's probability evolution. This emphasizes the inherent dynamism and complexity of non-trivial dialogues.

2. While individuals' final partitions tend to exhibit convergence towards elements in the joint partitions, they don't singularly converge to just one element. This indicates that even though there's some alignment in their reasoning, there remains a level of disagreement in how they derive their probabilities.

3. Regardless of the size of the partition, the number of iterations required for dialogues to approach convergence remains relatively stable. Our simulations consistently demonstrated that Bayesian

dialogues, on average, require around two iterations.

The algorithmic approach we employed was crucial in gleaning these insights. Through rigorous simulations across varied partition sizes, it shed light on the consistent nature of iterations and the nuances of convergence in Bayesian dialogues.

In essence, while non-trivial Bayesian dialogues exhibit a degree of unpredictability and complexity, there exist underlying patterns that remain consistent.

Bibliography

[1] Aumann, R. J. (1976). *"Agreeing to disagree."* The Annals of Statistics (4) : 1236–1239.

[2] Geanakoplos, J., & Polemarchakis, H. M. (1982). *"We can't disagree forever."* Journal of Economic Theory (28) : 192–200.

[3] Stam, A.J. (1983). *"Generation of a random partition of a finite set by an urn model."* Journal of Combinatorial Theory, Series A (35) : 231–240.

6 Appendix

Source Code

6.1 Appendix A (main algo)

```python
import numpy as np
import matplotlib.pyplot as plt
import argparse
import yaml

class posterior_probability_computation:
    def __init__(self, N, partition_1, partition_2, A, w):
        """Initialization for the class.
        Args:
            N (int): size of the universe.
            partition_1 (list): partition of the player alpha.
            partition_2 (list): partition of the player alpha.
            A (list): event we which to know the probability.
            w (int): event that occurs.
        """
        self.N = N
        self.partition_1 = partition_1
        self.partition_2 = partition_2
        self.A = A
        self.w = w

    def P(self, w, partition):
        """Assumption: w is a single element of the universe. P(w, partition) returns the
        set of partition w belongs to.
        Args:
            w (int): event that occurs.
            partition (list): arbitrary partition.
        Returns:
            set if w belongs to the set, None else.
        """
        for set in partition:
            if w in set:
                return set
        return None

    ### Useful functions

    def intersection(self, set_1, set_2):
        """Returns card(set_1 intersect set_2)"""
        list = []
        for i in set_1:
            if i in set_2:
                list.append(i)
        return list

    ### Step 1

    # set = P(w, partition)

    def q1_alpha(self, A, w, partition_1):
        """Return the posterior probability P(A | P(w, partition))."""
        P_alphaw = self.P(w, partition_1)
        return len(self.intersection(A, P_alphaw)) / len(P_alphaw), P_alphaw

    def find_sets0(self, A, w, partition_1, q1):
        """Return the sets of partition such that P(A | set) = proba (q0(A, w, partition))."""
        a_1 = []
        for k in range(len(partition_1)):
            set_k = partition_1[k]
            if len(self.intersection(A, set_k)) / len(set_k) == q1:
                a_1.append(set_k)
        return a_1

    def transforms_sets_of_sets(self, a_1):
        """Turns a set of partition into a single set with all elements.
        E.g., transforms_sets_of_sets([[1,2],[3,4]]) returns [1,2,3,4]"""
        list = []
        for i in range(len(a_1)):
            for e in a_1[i]:
                list.append(e)
        return list

    def q1_beta(self, A, w, partition_2, a_1_transformed):
        """Return the posterior probability of A for the second individual given its own information
        and the communication of the other player probability."""
        P_betaw = self.P(w, partition_2)
        numerator = len(self.intersection(self.intersection(A, P_betaw), a_1_transformed))
        denominator = len(self.intersection(P_betaw, a_1_transformed))
        return numerator / denominator, self.intersection(P_betaw, a_1_transformed)

    def find_sets1(self, A, w, a_1_transformed, partition_2, q1):
        """Return the sets of partition such that P(A | set, proba_other_player) = proba (q1(A, w, partition, a_1_transformed))."""
        b_1 = []
        for k in range(len(partition_2)):
            set_k = partition_2[k]
            numerator = len(self.intersection(self.intersection(A, set_k), a_1_transformed))
            denominator = len(self.intersection(set_k, a_1_transformed))
            if denominator != 0:
                if numerator / denominator == q1:
                    b_1.append(set_k)
        return b_1

    ### Step t

    def qt_alpha(self, A, w, partition_1, b_prec_transformed):
        """Return the posterior probability P(A | P(w, partition), b_prec) at time t."""
        P_alphaw = self.P(w, partition_1)
        return len(self.intersection(self.intersection(A, P_alphaw), b_prec_transformed)) / len(self.intersection(P_alphaw, b_prec_transformed)), self.intersectio

    def a_t(self, A, w, b_prec_transformed, partition_1, qt):
        """Return the sets of partition such that P(A | set, b_t_transformed) = proba (q1(A, w, partition))."""
        a_t = []
        for k in range(len(partition_1)):
            set_k = partition_1[k]
            numerator = len(self.intersection(self.intersection(A, set_k), b_prec_transformed))
            denominator = len(self.intersection(set_k, b_prec_transformed))
            if denominator != 0:
                if numerator / denominator == qt:
                    a_t.append(set_k)
        return a_t

    def qt_beta(self, A, w, partition_2, a_t_transformed):
        """Return the posterior probability of A for the second individual given its own information
        and the communication of the other player probability."""
        P_betaw = self.P(w, partition_2)
        numerator = len(self.intersection(self.intersection(A, P_betaw), a_t_transformed))
        denominator = len(self.intersection(P_betaw, a_t_transformed))
        return numerator / denominator, self.intersection(P_betaw, a_t_transformed)

    def b_t(self, A, w, a_t_transformed, partition_2, qt_beta):
        """Return the sets of partition such that P(A | set, proba_other_player) = proba (q1(A, w, partition, a_1_transformed))."""
        b_t = []
        for k in range(len(partition_2)):
            set_k = partition_2[k]
            numerator = len(self.intersection(self.intersection(A, set_k), a_t_transformed))
            denominator = len(self.intersection(set_k, a_t_transformed))
            if denominator != 0:
                if numerator / denominator == qt_beta:
                    b_t.append(set_k)
        return b_t

    def joint_partition(self, partition_1, partition_2):
        """Returns the joint partition of two partitions."""
        joint_partition_output = []
        for i in range(len(partition_1)):
            for j in range(len(partition_2)):
                if len(self.intersection(partition_1[i], partition_2[j])) > 0:
                    joint_partition_output.append(self.intersection(partition_1[i], partition_2[j]))
        return joint_partition_output

    def run(self):
        list_qt_alpha_proba = []
        list_qt_beta_proba = []
        self.join_initial_partition = self.joint_partition(self.partition_1, self.partition_2)

        # Run
        q1_alpha_proba, partition_ini_alpha = self.q1_alpha(self.A, self.w, self.partition_1)
        list_qt_alpha_proba.append(q1_alpha_proba)

        # Prior for beta computation
        elt_part_2 = self.P(self.w, self.partition_2)
        list_qt_beta_proba.append(len(self.intersection(self.A, elt_part_2)) / len(elt_part_2))

        # First step
        a_1 = self.find_sets0(self.A, self.w, self.partition_1, q1_alpha_proba)
        a_1_transformed = self.transforms_sets_of_sets(a_1)
        q1_beta_proba, partition_ini_beta = self.q1_beta(self.A, self.w, self.partition_2, a_1_transformed)
        self.partition_1 = a_1
        b_1 = self.find_sets1(self.A, self.w, a_1_transformed, self.partition_2, q1_beta_proba)
        b_1_transformed = self.transforms_sets_of_sets(b_1)
        self.partition_2 = b_1

        qt_alpha_proba = q1_alpha_proba
        qt_beta_proba = q1_beta_proba
        b_before = b_1_transformed

        list_qt_alpha_proba.append(qt_alpha_proba)
        list_qt_beta_proba.append(qt_beta_proba)

        # Second step
        qt_alpha_proba, partition_alpha = self.qt_alpha(self.A, self.w, self.partition_1, b_before)
        at = self.a_t(self.A, self.w, b_before, self.partition_1, qt_alpha_proba)
        a_t_transformed = self.transforms_sets_of_sets(at)
        self.partition_1 = at
        qt_beta_proba, partition_beta = self.qt_beta(self.A, self.w, self.partition_2, a_t_transformed)
        bt = self.b_t(self.A, self.w, a_t_transformed, self.partition_2, qt_beta_proba)
        b_before = self.transforms_sets_of_sets(bt)
        self.partition_2 = bt
        b_before_before = b_1_transformed
        a_t_transformed_before = a_1_transformed

        # All next steps
        while b_before != b_before_before and a_t_transformed_before != a_t_transformed:
            b_before_before = b_before
            a_t_transformed_before = a_t_transformed
            list_qt_alpha_proba.append(qt_alpha_proba)
            list_qt_beta_proba.append(qt_beta_proba)

            qt_alpha_proba, partition_alpha = self.qt_alpha(self.A, self.w, self.partition_1, b_before)
            at = self.a_t(self.A, self.w, b_before, self.partition_1, qt_alpha_proba)
            a_t_transformed = self.transforms_sets_of_sets(at)
            self.partition_1 = at
            qt_beta_proba, partition_beta = self.qt_beta(self.A, self.w, self.partition_2, a_t_transformed)
            bt = self.b_t(self.A, self.w, a_t_transformed, self.partition_2, qt_beta_proba)
            self.partition_2 = bt
            b_before = self.transforms_sets_of_sets(bt)

        list_qt_alpha_proba.append(qt_alpha_proba)
        list_qt_beta_proba.append(qt_beta_proba)

        print(list_qt_alpha_proba, list_qt_beta_proba)
        return list_qt_alpha_proba, list_qt_beta_proba, len(self.joint_partition([partition_alpha], self.join_initial_partition))

    def visualisations(self, list_qt_alpha_proba, list_qt_beta_proba):
        Time_steps = np.arange(0, len(list_qt_alpha_proba))
        plt.figure(figsize=(5, 5))
        plt.grid()
        plt.plot(Time_steps, list_qt_alpha_proba, label='alpha')
        plt.plot(Time_steps, list_qt_beta_proba, label='beta')
        plt.xlabel('Iteration␣number')
        plt.ylabel('Posterior␣probabilities')
        plt.xticks(np.arange(0, len(list_qt_alpha_proba), step=1))
        plt.legend()
        plt.show()
        plt.close()

### Main
def main_run(N, partition_1, partition_2, A, w, visualisations=True):
    my_experiment = posterior_probability_computation(N=N, partition_1=partition_1, partition_2=partition_2, A=A, w=w)
    list_qt_alpha_proba, list_qt_beta_proba, length = my_experiment.run()
    if visualisations:
        my_experiment.visualisations(list_qt_alpha_proba, list_qt_beta_proba)
    return length

if __name__ == '__main__':
    with open('config.yaml', 'r') as yaml_file:
        data = yaml.safe_load(yaml_file)
    main_run(N=data['N'], partition_1=data['partition_1'], partition_2=data['partition_2'], A=data['A'], w=data['w'], visualisations=True)
```

6.2 Appendix B (Code for tests)

```
import numpy as np
import matplotlib.pyplot as plt
import argparse
import yaml
from run import main_run, posterior_probability_computation
def generate_random_A(N) :
  """ Generates a random list of elements in {1,..,N} with random size in {1,...,N}."""
  length_A = np.random.randint(1,N+1)
  A = []
  while len(A)!=length_A:
    a = np.random.randint(1,N+1)
    if a not in A :
      A.append(a)
  return A

def T(n, N=15):
    # Bell number (N=25 ?)
    total_sum = 0
    for k in range(1, N):
        total_sum += k ** n / np.math.factorial(k)
    return total_sum / np.exp(1)

def define_discrete_dist(u, n):
    # Discrete distribution
    return np.exp(-1) * (u ** n) / np.math.factorial(u)

def sample_discrete_dist(n, u_max=10):
    # Sampling of the number of urns
    density = [define_discrete_dist(u, n) for u in range(u_max)]
    return np.random.choice(a=np.arange(u_max), p=[e / sum(density) for e in density])

def sample_random_partition(n, u_max):
    u = sample_discrete_dist(n, u_max)
    partition = [[] for _ in range(u)]
    for b in range(1, n + 1):
        urn_chosen = np.random.randint(1, u + 1)
        partition[urn_chosen - 1].append(b)
    # Remove empty subsets from the partition
    partition = [subset for subset in partition if subset]
    return partition

def permutation_inside_sets(partition, seed=123) :
  """Permutes randomly the order of the sets elements in a partition."""
  np.random.seed(seed) # Set the seed.
  perm_partition = []
  for set in partition :
    n = len(set)
    perm_set = permutation_list(set)
    perm_partition.append(perm_set)
  return perm_partition

def permutation_list(list,seed = 123) :
  """Returns a random permutation of the elements of the list."""
  np.random.seed(seed) # Set the seed.
  n = len(list)
  permuted_list = []
  indices_permuted = np.random.permutation(n)
```

```python

  for i in indices_permuted :
    permuted_list.append(list[i])

  return permuted_list

def tests_different_seeds(n_tests,N) :
  """ Test if the code runs with random arguments."""

  length = []

  for test_number in range(n_tests) :

    np.random.seed(test_number) # Set the seed.
    partition_1 = sample_random_partition(N,N+1)
    partition_2 = sample_random_partition(N,N+1)
    A = generate_random_A(N)
    w = A[np.random.randint(0,len(A))]
    length.append(main_run(N, partition_1, partition_2, A, w, visualisations=False))
    print({'partition_1:␣{}'.format(partition_1), 'partition_2:␣{}'.format(partition_2),
           'A:␣{}'.format(A), 'w:␣{}'.format(w)})

  length = np.array(length)
  print(f"␣n_tests␣{n_tests}")
  print(f"␣min␣{np.min(length)}")
  print(f"␣max␣{np.max(length)}")

def all_items_equality_check(dict) :
  first = True
  for key in dict.keys() :
    if first :
      first_value = dict[key]
      print(first_value)
      first = False
    else :
      value = dict[key]
      print(value)
      assert value == first_value

def tests_invariance_seeds(n_tests,N,seed=501) :
  """ Test if the code runs with random arguments."""

  seed = seed
  np.random.seed(seed)
  A = generate_random_A(N)
  w = A[np.random.randint(0,len(A))]
  partition_1 = sample_random_partition(N,N+1)
  partition_2 = sample_random_partition(N,N+1)

  results_runs = {}

  for test_number in range(n_tests) :

    seed = test_number+1
    perm_partition_1 = permutation_list(partition_1,seed)
    perm_partition_2 = permutation_list(partition_2,seed)

    my_experiment = posterior_probability_computation(N=N,partition_1=perm_partition_1,partition_2=perm_partition_2,A=A,w=w)
    list_qt_alpha_proba, list_qt_beta_proba = my_experiment.run()
    results_runs['{}'.format(test_number)] = (list_qt_alpha_proba, list_qt_beta_proba)

  all_items_equality_check(results_runs)

  print({'partition_1:␣{}'.format(partition_1), 'partition_2:␣{}'.format(partition_2),
          'A:␣{}'.format(A), 'w:␣{}'.format(w)})

def tests_invariance_inside_permutation(n_tests,N,ini_seed) :
  """ Test if the code runs with random arguments."""

  np.random.seed(ini_seed)
  A = generate_random_A(N)
  w = A[np.random.randint(0,len(A))]
  partition_1 = sample_random_partition(N,N+1)
  partition_2 = sample_random_partition(N,N+1)

  results_runs = {}

  for test_number in range(n_tests) :

    seed = test_number+1
    perm_partition_1 = permutation_list(partition_1,seed)
    perm_partition_2 = permutation_list(partition_2,seed)

    perm_partition_1 = permutation_inside_sets(perm_partition_1, seed)
    perm_partition_2 = permutation_inside_sets(perm_partition_2, seed)

    my_experiment = posterior_probability_computation(N=N,partition_1=perm_partition_1,partition_2=perm_partition_2,A=A,w=w)
    list_qt_alpha_proba, list_qt_beta_proba = my_experiment.run()
    results_runs['{}'.format(test_number)] = (list_qt_alpha_proba, list_qt_beta_proba)

  all_items_equality_check(results_runs)

  print({'partition_1:␣{}'.format(partition_1), 'partition_2:␣{}'.format(partition_2),
          'A:␣{}'.format(A), 'w:␣{}'.format(w)})

if __name__=='__main__':
  n_tests = 10
  N = 10
  # tests_invariance_inside_permutation(n_tests,N,ini_seed=1334)
  tests_different_seeds(n_tests,N)
```