Management Science

Working Paper 2019:4

BinaryBandit: An Efficient Julia Package for Optimization and Evaluation of the Finite-Horizon Bandit Problem with Binary Responses

Peter Jacko

The Department of Management Science
Lancaster University Management School
Lancaster LA1 4YX
UK

BinaryBandit: An efficient Julia package for optimization and evaluation of the finite-horizon bandit problem with binary responses

Peter Jacko
Department of Management Science
Lancaster University, UK

August 23, 2019

Abstract

Variants of the multi-armed bandit problem for design of sequential experiments have been studied in several disciplines for almost a century, but the performance evaluation of proposed designs or finding a Bayes-optimal design over the finite horizon has resisted derivation of a closed formulae. Computational optimization and evaluation is thus the only possible approach. The BinaryBandit package in Julia programming language has been developed to provide such framework with a number of designs, easily extendable to add new designs. The package is based on the use an efficient implementation of backward recursion which gives accurate (up to computer accuracy) evaluation for small and moderate horizons. For instance, on a standard laptop or desktop computer, the Bayes-optimal design for the two-armed problem can be computed for offline use or evaluated in online fashion in a few minutes (horizon around $1,000$), in a few hours (horizon around $2,000$), or in a few days (horizon around $4,000$). 32GB of RAM allows storing (e.g., for offline use) of the whole design up to horizon around 1440; when its storing is not needed (e.g., for Bayesian evaluation or for calculation of the initial action) it allows up to horizon around 4440. These problems are significantly larger than what has been reported in the literature, since moderate and large horizons have only been evaluated by simulation, trading-off accuracy. This paper describes the details of the backward recursion implementation and gives an example of the package usage.

Keywords: multi-armed bandit problem, design of sequential experiments, Bayesian decision theory, dynamic programming, index rules, response-adaptive randomization, Julia.

1. Introduction

The multi-armed bandit problem represents in a simplified way the general question of how we learn—or should learn—from past experience. It is a model of allocation of sequentially arriving subjects to the available interventions (arms) with unknown rewards in order to resolve the trade-off between earning of rewards from arms that have performed well so far and learning about the mean rewards with the aim to identify the highest-rewarding arm. Because such situations are omnipresent, such problems have been studied in several disciplines, e.g., biostatistics, operations research, reinforcement learning, economics, telecommunications, marketing, etc., with a variety of different weights on the objectives of earning and learning, and with different features arising from practice.

A formulation of the problem using the Bayesian decision-theoretic framework allows for Bayes-optimality. Practical application of this Bayesian approach has however been long hindered by its computational complexity, since the optimal solution is known in analytical form only for infinite horizon (Kelly 1981). A variety of practical approximations and heuristics have been developed and studied across several disciplines in order to overcome this issue, but their analysis failed to give exact results or bounds sufficiently close to Bayes-optimality for finite horizon problems, which are the problems most relevant to many situations in practice.

In this paper we thus focus on the finite-horizon setting. We present a Julia package called **BinaryBandit**[1] which assumes that subject responses are binary (success/failure), motivated by its widespread applicability and by being one of the most studied settings. We also restrict the discussion to two arms, which often naturally appears per se or as a subproblem in some multi-armed generalizations (e.g., if new arms appear over time), and serves as a starting point for introducing additional problem features. See, e.g., Jacko (2019) for a multidisciplinary survey in which this package has been used to present computational results in such a setting.

Future versions of the package are planned to include also settings with more than two arms. Other packages are planned to be developed using similar ideas for responses other than binary.

In this paper we describe the framework for computing the Bayes-optimal (a.k.a. Bayesian decision-theoretic) design using backward recursion and its efficient algorithmic implementation in the **BinaryBandit** package. The package also includes functions for evaluation of a variety of Bayesian and non-Bayesian designs using algorithms that are straightforwardly adapted from this one, and are not described here in detail.

2. Related Software

To the best of our knowledge, there are no publicly available packages in any software that would compute the Bayes-optimal design. This is commonly believed to be computationally intractable for practical problems (see Jacko 2019, Section 7.1 Myth #1). Table 1 lists academic literature in which this design was reported, indicating that the authors were typically limited to the horizon around 100, some of them mentioning that horizons beyond around 200 are intractable.

There are related packages devoted to the multi-armed bandit problem, e.g.,

BanditsBook: using simulation, it evaluates several designs, including UCB, ϵ-greedy, softmax and several of their variants; packages are provided in Python, Julia, R and Ruby, with links to packages in several other programming languages. `https://github.com/johnmyleswhite/BanditsBook`

pymaBandits: using simulation, it evaluates several designs, including UCB, Gittins index rule, Thompson sampling and several of their variants, for binary, Poisson and Exponential responses; packages are provided in Matlab and Python. `https://mloss.org/software/view/415/`

google/MAB: using simulation, it evaluates several designs, including UCB, ϵ-greedy, Thomp-

[1]The package can be found at `https://github.com/PeterJacko/BinaryBandit`.

Publication	T	$T^{\max}$	SW, HW, RAM
Steck (1964)	25	N/A	N/A, UNIVAC 1105, 54 kB
Yakowitz (1969)	5	N/A	Fortran, N/A, N/A
Berry (1978)	100	N/A	Basic (?), Atari (?), N/A
Hardwick (1991)	160	160[2]	N/A, N/A, N/A
Ginebra and Clayton (1999)	150	180	N/A, N/A, N/A
Hardwick, Oehmke, and Stout (2006)[3]	100	200	N/A, N/A, N/A
Ahuja and Birge (2016)[4]	96	240	N/A, Mac 4GB
Williamson, Jacko, Villar, and Jaki (2017)	100	215	R, PC, 16GB
Villar (2018)	100	N/A	Matlab, PC, N/A
Kaufmann (2018)	70	N/A	N/A, N/A, N/A
This paper	4440	4440	Julia 1.0.1 & BB, PC, 32GB

Table 1: Horizons T with reported results and $T^{\max}$ reported as the largest computationally tractable on a standard computer by backward recursion in the literature on two-armed problem with binary responses.

son sampling and several of their variants, for binary, Poisson and Gaussian responses; package is provided in R. `https://github.com/google/MAB`

bandit: it provides commands useful to evaluate the 1:1 design and Thompson sampling for binary and Poisson responses; package is provided in R. `https://cran.r-project.org/web/packages/bandit/bandit.pdf`

contextual: is focusses on several generalizations of the bandit problem, but for our basic setting, using simulation, it evaluates several designs, including UCB, ϵ-greedy, Thompson sampling and several other, for binary and Gaussian responses; package is provided in R. `https://arxiv.org/pdf/1811.01926.pdf`

banditlib: using simulation, it evaluates several designs, including UCB, ϵ-greedy, Thompson sampling and several of their variants, for binary and Gaussian responses; package is provided in C++. `https://github.com/jkomiyama/banditlib`

More packages (less related to ours) are described in `https://arxiv.org/pdf/1811.01926.pdf` and `https://en.wikipedia.org/wiki/Multi-armed_bandit`.

3. Model

In this section we briefly describe the two-armed problem with binary responses as a Markov decision process. See Jacko (2019, Section 3) for a more detailed formulation.

We consider arms labelled by $k \in \mathcal{K} := \{C, D\}$. The response set is denoted by $\mathcal{O} := \{0, 1\}$. Subject responses are uncertain, i.e., modelled as Bernoulli-distributed with parameter $0 \leq \theta_k \leq 1$, the *success probability*, independent across arms. The responses are *immediate*. Subjects arrive at time epochs $t \in \mathcal{T} := \{0, 1, 2, \ldots, T-1\}$, where $T \leq +\infty$ is the *time horizon*.

[2] $T^{\max} = 320$ was reported using a parallelized algorithm on the CNSF's IBM 3090 with parallel facilities.
[3] This paper considered a generalized model with delayed responses.
[4] This paper considered a generalized model with subjects allocated in groups.

States. The *physical state* is represented by the numbers of observed successes and failures on arm C denoted by s_C and f_C, respectively, and the numbers of observed successes and failures on arm D denoted by s_D and f_D, respectively, which we denote by $\boldsymbol{x} := (s_C, f_C, s_D, f_D)$. Note that at time epoch t, $s_C + f_C + s_D + f_D = t$. In addition to the physical state, there is an *information state*, the parameters of the prior distributions for the success probability of each arm, which does not change during the trial, thus we omit it from notation.

The prior distribution for arm k is the Beta distribution with parameters $\widetilde{s}_k(0), \widetilde{f}_k(0)$. These parameters can be interpreted as the numbers of pseudo-observations of successes and failures before the start of the trial. The conventional assumption is to take the uniform distribution as a prior distribution on each arm k, i.e., $(\widetilde{s}_k(0), \widetilde{f}_k(0)) = (1, 1)$.

Actions. At every time epoch $t \in \mathcal{T}$ the design must prescribe how the arrived subject should be allocated to interventions. We consider an action set of two *pure actions*, which we call action 1 (allocating to arm C) and action 2 (allocating to arm D). For convenience in situations when both actions are optimal, we also consider an *equally-weighted mixed action*, which we call action 3 (allocating to either arm with probability 1/2). Formally, the action set is $\mathcal{A} = \{1, 2, 3\}$. At time epoch T there is no decision to be made, which can be achieved by setting $1 \equiv 2 \equiv 3$.

Transition Probabilities. At every time epoch $t \in \mathcal{T}$, in state $\boldsymbol{x} = (s_C, f_C, s_D, f_D)$, each arm k has the posterior distribution given, because of conjugacy, by the Beta distribution with parameters $\widetilde{s}_k, \widetilde{f}_k$, briefly $\text{Beta}(\widetilde{s}_k, \widetilde{f}_k)$, where $\widetilde{s}_k = \widetilde{s}_k(0) + s_k$ and $\widetilde{f}_k = \widetilde{f}_k(0) + f_k$. The probability $q_{k,\boldsymbol{x},o}$ of observing response $o \in \mathcal{O}$ for the current subject if it is allocated to arm $k \in \mathcal{K}$ in state $\boldsymbol{x}$ thus is

$$q_{k,\boldsymbol{x},1} = \frac{\widetilde{s}_k}{\widetilde{s}_k + \widetilde{f}_k} \qquad\qquad q_{k,\boldsymbol{x},0} = \frac{\widetilde{f}_k}{\widetilde{s}_k + \widetilde{f}_k}$$

The (non-zero) transition probabilities $h^a_{\boldsymbol{x},\boldsymbol{x}'}$ of moving from state $\boldsymbol{x}$ to state $\boldsymbol{x}'$ under action a are

$$h^1_{\boldsymbol{x},\boldsymbol{x}'} = \begin{cases} q_{C,\boldsymbol{x},1} & \text{if } \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{e}_1 \\ q_{C,\boldsymbol{x},0} & \text{if } \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{e}_2 \end{cases} \qquad h^2_{\boldsymbol{x},\boldsymbol{x}'} = \begin{cases} q_{D,\boldsymbol{x},1} & \text{if } \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{e}_3 \\ q_{D,\boldsymbol{x},0} & \text{if } \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{e}_4 \end{cases}$$

where $\boldsymbol{e}_j$ is the standard basis vector.

Rewards. The expected one-period reward $r^a_{\boldsymbol{x}}$ for all states $\boldsymbol{x}$ and all actions a is as follows: $r^a_{\boldsymbol{x}} = 0$ for all states such that $s_C + f_C + s_D + f_D \le T - 1$ and $r^a_{\boldsymbol{x}} = s_C + s_D$ for all states such that $s_C + f_C + s_D + f_D = T$ (i.e., the reward is the number of observed successes in all states in which the trial can eventually end).

State and Action Processes. The evolution of a Markov decision process is captured by the state process $\boldsymbol{X}(\cdot)$, and the action process which depends on the state process, but can be briefly written as $A(\cdot)$, where $A_{\boldsymbol{X}(t)} \in \mathcal{A}$.

Designs. A particular design (policy) π prescribes the action process $A(\cdot)$. Let Π be the set of designs that are non-anticipating and satisfy the above constraints on $A(\cdot)$. Let us denote by $\mathbb{E}_t^\pi[\cdot]$ the Bayes expectation under design $\pi \in \Pi$ conditioned on information available at time epoch $t \in \mathcal{T}$.

Objective. We focus on the *Bayes number of successes* as the principal performance measure, which is

$$\mathbb{N}_t^\pi(\boldsymbol{x}) := \mathbb{E}_t^\pi \left[\sum_{u=t}^{T} r_{\boldsymbol{X}(u)}^{A_{\boldsymbol{X}(u)}} \middle| \boldsymbol{X}(t) = \boldsymbol{x} \right]. \tag{1}$$

The objective is to find an optimal design π^* that maximises the mean Bayes number of successes as evaluated at time epoch $t = 0$ when there are no observations ($\boldsymbol{x} = \boldsymbol{0}$), i.e.,

$$\pi^* := \arg\max_{\pi \in \Pi} \mathbb{N}_0^\pi(\boldsymbol{0}). \tag{2}$$

Instead of the Bayes number of successes, we report two equivalent measures: the *Bayes proportion of successes* and the *Bayes regret number of successes*, as these provide complementary interpretation and insights. The mean Bayes proportion of successes is

$$\mathbb{P}_t^\pi(\boldsymbol{x}) := \frac{1}{T-t} \mathbb{N}_t^\pi(\boldsymbol{x}). \tag{3}$$

We further define the mean Bayes regret number of successes,

$$\mathbb{R}_t^\pi(\boldsymbol{x}) = \mathbb{E}\left[\max\{\theta_C, \theta_D\}\right] - \mathbb{N}_t^\pi(\boldsymbol{x}). \tag{4}$$

For a problem with uniform distribution as a prior distribution on each arm, as considered in this paper, $\mathbb{E}\left[\max\{\theta_C, \theta_D\}\right] = 2/3$. Note that all the three measures depend on parameter T, although we have suppressed the explicit notation.

4. Computation of the Bayes-Optimal Design

The well-known Bellman's principle of optimality for dynamic programming states that this problem can be solved recurrently backwards in time, by the so-called *backward recursion* algorithm, which implicitly finds the family of optimal designs starting from all possible system states at all time epochs, as these altogether form the optimal design π^*. The optimal design thus also gives $\mathbb{N}_t^{\pi^*}\left((s_k, f_k)_{k\in\mathcal{K}}\right)$ which we will briefly denote by $\mathbb{N}_t^*\left((s_k, f_k)_{k\in\mathcal{K}}\right)$, and which for all possible system states $(s_k, f_k)_{k\in\mathcal{K}}$ satisfies the system of *Bellman equations*

$$\mathbb{N}_t^*(\boldsymbol{x}) = \max_{a \in \mathcal{A}} \left\{ r_{\boldsymbol{x}}^a + \sum_{\boldsymbol{x}' \in \mathcal{X}} h_{\boldsymbol{x},\boldsymbol{x}'}^a \mathbb{N}_{t+1}^*(\boldsymbol{x}') \right\} \qquad \text{for } t \in \mathcal{T} \tag{5}$$

$$\mathbb{N}_T^*(\boldsymbol{x}) = \max_{a \in \mathcal{A}} r_{\boldsymbol{x}}^a \tag{6}$$

which after plugging the definitions of rewards and transition probabilities gives

$$\begin{aligned}\mathbb{N}_t^*(s_C, f_C, s_D, f_D) = \max_{\ell\in\mathcal{K}} \Bigg\{ & \frac{\widetilde{s}_\ell}{\widetilde{s}_\ell + \widetilde{f}_\ell}\mathbb{N}_{t+1}^*(s_C + \delta_{C,\ell}, f_C, s_D + \delta_{D,\ell}, f_D) \\ & + \frac{\widetilde{f}_\ell}{\widetilde{s}_\ell + \widetilde{f}_\ell}\mathbb{N}_{t+1}^*(s_C, f_C + \delta_{C,\ell}, s_D, f_D + \delta_{D,\ell}) \Bigg\} \quad \text{for } t \in \mathcal{T} \qquad (7) \\ \mathbb{N}_T^*(s_C, f_C, s_D, f_D) = s_C + s_D & \qquad (8)\end{aligned}$$

where action $\ell \in \mathcal{K}$ means allocating the $(t+1)$-st subject to arm ℓ and where the Kronecker delta $\delta_{k,\ell}$ is used to update the posterior distribution with the response of that subject.

We will denote by $a_t^*(s_C, f_C, s_D, f_D)$ the optimal action obtained from the above equations by taking $\arg\max$ instead of max operator, where $\arg\max$ returns action 3 in case of a tie between actions 1 and 2.

The backward recursion algorithm starts by enumerating all the possible states in the final time epoch, continuing backwards in time while employing the Bellman equation in every state.

4.1. Achieving Computational Efficiency Using BinaryBandit Package

Note that both $\mathbb{N}_t^*(s_C, f_C, s_D, f_D)$ and $a_t^*(s_C, f_C, s_D, f_D)$ are 5-dimensional. Each of t, s_C, f_C, s_D, f_D can assume an integer value between 0 and T (i.e. $T+1$ values), i.e., storing them in an array requires $(T+1)^5$ elements. If each element takes the today's software variables standard size of 64 bits, such array takes around 74.5GB for $T = 99$ and around 7.1PB for $T = 999$.

The possible states s_C, f_C, s_D, f_D at time epoch $t \in \mathcal{T}$ satisfy the conservation law $s_C + f_C + s_D + f_D = t$, meaning that the total number of observations at time epoch t is t. The fifth dimension thus can be calculated from knowing the other four, which allows using 4-dimensional arrays, which take around 762MB for $T = 99$ and around 7.3TB for $T = 999$.

In such 4-dimensional arrays, only the elements satisfying $s_C + f_C + s_D + f_D \leq T$ are relevant, which take only around 4% of all the elements (except for very small values of T). More precisely, there is $\binom{T+4}{4} = (T+4)(T+3)(T+2)(T+1)/24$ of such relevant elements. In the **BinaryBandit** package we thus store these in vectors (1-dimensional arrays) to enumerate only the relevant states and use a *storage mapping function* to locate states. These take around 33.7MB for $T = 99$ and around 312.3GB for $T = 999$.

Additional reduction can be achieved if it is not required to output $\mathbb{N}_t^*(\boldsymbol{x})$ for all the states and time epochs. Indeed, package users are often interested only in $\mathbb{N}_0^*(\mathbf{0})$ and $a_0^*(\mathbf{0})$, which we refer to as the *online calculation*. It is obvious from the system of Bellman equations that we only need to access $\mathbb{N}_{t+1}^*$ for computing $\mathbb{N}_t^*$, so $\mathbb{N}_s^*$ for $s > t+1$ do not need to be stored during the computation. It is also clear that computing $\mathbb{N}_t^*(\boldsymbol{x})$ only requires access to $\mathbb{N}_{t+1}^*(\boldsymbol{x}')$ where $\boldsymbol{x}' > \boldsymbol{x}$, so if the algorithm, when computing $\mathbb{N}_t^*$, loops through states in a carefully thought order, the newly computed values can be stored by rewriting $\mathbb{N}_{t+1}^*$. We can thus drop the time epoch subscript and write $\mathbb{N}^*(\boldsymbol{x})$ for the value-to-go vector during the computation. See Algorithm 1 for an algorithmic scheme for computing $\mathbb{N}_0^*(\mathbf{0})$. Note that computing the value-to-go vector requires the storage mapping function whose details are not explained here (the formulae can be found in the code). Vector $\mathbb{N}^*(\boldsymbol{x})$ has $\binom{T+3}{3} = (T+3)(T+2)(T+1)/6$ elements in the loop corresponding to time epoch $t = T$ (it gets smaller in every loop step,

Algorithm 1 Algorithmic scheme for computing $\mathbb{N}^*_0(\mathbf{0})$

```
INPUT: $T$
state_index = 0
for $s_D = 0 : T, f_C = 0 : (T - s_D), s_C = 0 : (T - s_D - f_C)$ do
  state_index += 1
  $\mathbb{N}^*$[state_index] = value computed using (8)
end for
for $t = (T - 1) : -1 : 0$ do
  state_index = 0
  for $s_D = 0 : t, f_C = 0 : (t - s_D), s_C = 0 : (t - s_D - f_C)$ do
    state_index += 1
    $\mathbb{N}^*$[state_index] = value computed using (7)
  end for
end for
OUTPUT: $\mathbb{N}^*[1]$
```

but it is typically not runtime-efficient nor necessary to change its size dynamically) and takes around 1.3MB for $T = 99$ and around 1.2GB for $T = 999$ using the Float64 type. Using the Float32 type it takes a half of these values, and we have found Float32 to provide a good trade-off for practice between accuracy (theoretically, the accuracy is at least 4 significant digits) and size of the problem ($T = 3720$ takes around 32GB which is the RAM standard on performance desktop and laptop computers today, and even larger horizons can be computed thanks automatic memory swapping although with a notable increase in runtime). In terms of runtime, `Julia` is fast in processing `for` loops as long as arrays are accessed along columns as these are stored in column-major order. To speed up the code, we also use the `@inbounds` macro which removes array index out-of-bounds checks.

Figure 1 illustrates the computational complexity of online calculation of the Bayes-optimal design using the function `DP_2_action_lin` of the **BinaryBandit** package with argument `float_version = 32` in order to set the stored value-to-go type to Float32. Note that to improve accuracy, calculations of the Bellman equation are done using Float64 type for the rewards and transition probabilities. It may be important to note that in the number of elements (i.e., without multiplying by the memory required for each element), the memory requirement presented in Figure 1(right) is 2 times larger than that of Gluss (1962) because of his elimination of symmetric states, which can be done if the prior Beta distributions are the same for the two arms. Although the idea is not new, its implementation using storage mapping function is rare in the high-level scientific software of today.

The computational complexity of *offline calculation* (i.e., outputting the whole design, i.e., the optimal actions of all possible states) of this design is very similar to online calculation in terms of the runtime, but radically different in terms of memory requirement. The array of optimal actions cannot be reduced to 3 dimensions as we did with the value-to-go. Using the storage mapping function we have $\binom{T+4}{4} = (T+4)(T+3)(T+2)(T+1)/24$ relevant elements in the action vector. Note however that the actions that need to be stored only take values from $\mathcal{A}$, i.e., $1, 2, 3$, which can be stored using 2 bits. In the **BinaryBandit** package we thus store the action vector by encoding every 4 consecutive actions in a single value of UInt8 type.

The offline calculation of the Bayes-optimal design is done using the function `DP_2_policy_bin_lin`

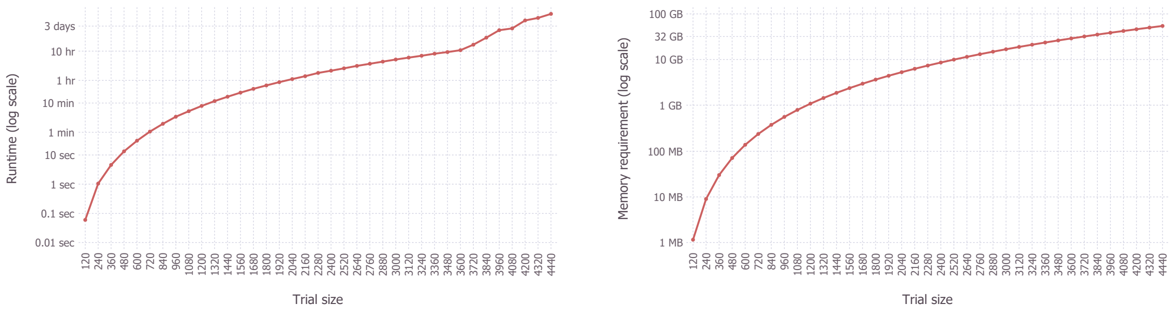

Figure 1: An illustration of computational complexity of online calculation of the Bayes-optimal design over a range of trial sizes.

Software	RAM	$T = 60$	$T = 120$	$T = 180$	$T = 240$	$T = 300$	$T^{\max}$
Julia 0.6.2 & ad hoc	12 GB	2sec	22sec	108sec	331sec	789sec	420
Julia 1.0.1 & ad hoc	12 GB	1sec	17sec	82sec	262sec	643sec	420
R & ad hoc	16 GB	1sec	12sec	59sec	191sec	N/A	240
Julia 1.0.1 & BB	31 GB	0.0036sec	0.046sec	0.23sec	0.73sec	1.6sec	1440
R & ad hoc	5 GB	1sec	6sec	26sec	84sec	209sec	420
Julia 1.0.1 & BB	31 GB	0.0040sec	0.056sec	0.27sec	0.91sec	2.8sec	4440

Table 2: A comparison of runtime of the Bayes-optimal design for $T = 60 : 60 : 300$ and the largest horizon $T^{\max}$ (as a multiple of 60) which does not give an out-of-memory error using code implementations by author's colleagues and students. The top four are for offline calculation, the bottom two are for online calculation. BB refers to the use of the BinaryBandit package.

of the *BinaryBandit* package. The standard Float64 type is used for the value-to-go vector as the memory reduction by using the Float32 type does not help significantly as the action vector is approximately $T/4$ times larger. A computer with 32GB RAM is able to solve the problem of up to trial size $T = 1,440$, keeping all the optimal actions in RAM during the calculations. For practical purposes, however, it may be possible to store parts of the solution on hard disk and thus relieve RAM memory to allow calculation for larger trial sizes.

A number of author's colleagues and PhD students who had programmed this design for their needs and have kindly provided runtimes of their code implementations, which are presented in Table 2 for comparison, indicating that the BinaryBandit package is two orders of magnitude faster than ad hoc codes and is able to solve a few times larger problems.

Algorithm 2 gives a simple example of use of the two above-mentioned functions.

5. Summary and Discussion

In this paper we have briefly presented the **BinaryBandit** package and illustrated the way in which the computation of the Bayes-optimal design is implemented. Despite the algorithmic scheme or the code may look very simple, it is a result of a considerate effort over the past two years of finding an implementation which is both efficient in terms of computational resources

Algorithm 2 Example code

```
using BinaryBandit
T = 100
( DP_2_online , DP_2_value_online ) = DP_2_action_lin( T , 32 )
( DP_2_offline , DP_2_value_offline ) = DP_2_policy_bin_lin( T )
```

and short in terms of the number of lines of code. Besides the two recommended functions for the Bayes-optimal design presented here, the package currently implements other 8 variants of these functions (which are slower or less memory-efficient). For each of these, the package also has a corresponding variant allowing for user-defined final-period rewards. There is around 1000 lines of code for this design. In addition, there is a similar amount of code for the more general constrained randomized Bayes-optimal design introduced in Williamson *et al.* (2017).

Besides optimization, the package also contains modules for evaluation of the Bayes optimal and of other 12 designs, including the Myopic, UCB, Knowledge gradient, and other designs. This evaluation is in the frequentist sense, assuming that the success probabilities θ_C, θ_D are known. It is implemented using dynamic programming (replacing the max operator of the Bellman equation by the respective action prescribed by a design) instead of by simulation, yielding a perfectly accurate evaluation (subject to computational accuracy of the chosen numerical type). Current performance evaluation measures include the mean and the standard deviation of the number of successes, but work is in progress to additionally incorporate other measures of interest such as the bias of the maximum-likelihood estimator, its standard error, and measures (a.k.a. operating characteristics) related to hypothesis testing.

Next versions of the package are planned to include functions for Bayes-optimization and for evaluation of bandit problems with more than two arms, such as the Gittins and Whittle index rules, and to cover randomized designs such as those common in the biostatistics literature for adaptive clinical trials. The package will be openly published (expected by the end of July 2019) and fully available for corrections, performance improvement and additions by other developers.

As such the **BinaryBandit** package is useful both for practitioners to evaluate, compare and implement existing designs, and for researchers to compare the performance of new design against the existing ones.

Computational Details

The results in this paper were obtained using Julia 1.0.1, distributed as JuliaPro, with the **FileIO**, **Printf**, **Gadfly**, and **Cairo** packages. Julia itself and all packages used are available from `https://julialang.org/` and `https://juliacomputing.com/products/juliapro`.

Acknowledgments

The author is grateful to James Edwards and Livia Stark who have provided the data presented in Table 2 and to his colleagues and PhD students who attended his seminar and gave valuable feedback on the early versions of the package and the paper. This package is being developed by the Group on Optimal Adaptive Learning (G.O.A.L.) at Lancaster University,

UK (see http://www.lancaster.ac.uk/staff/jacko/goal/).

References

Ahuja V, Birge JR (2016). "Response-adaptive Designs for Clinical Trials: Simultaneous Learning from Multiple Patients." *European Journal of Operational Research*, **248**, 619–633.

Berry DA (1978). "Modified Two-Armed Bandit Strategies for Certain Clinical Trials." *Journal of the American Statistical Association*, **73**, 339–345.

Ginebra J, Clayton MK (1999). "Small-Sample Performance of Bernoulli Two-Armed Bandit Bayesian Strategies." *Journal of Statistical Planning and Inference*, **79**(1), 107–122.

Gluss B (1962). "A Note on a Computational Approximation to the Two-Machine Problem." *Information and Control*, **5**(3), 268–275.

Hardwick J, Oehmke R, Stout QF (2006). "New Adaptive Designs for Delayed Response Models." *Journal of Statistical Planning and Inference*, **136**, 1940–1955.

Hardwick JP (1991). "Computational Problems Associated with Minimizing the Risk in a Simple Clinical Trial." *Contemporary Mathematics: Statistical Multiple Integration*, **115**, 239–256.

Jacko P (2019). "The Finite-Horizon Two-Armed Bandit Problem with Binary Responses: A Multidisciplinary Survey of the History, State of the Art, and Myths." *Management Science Working Paper 2019:3*, Lancaster University Management School.

Kaufmann E (2018). "On Bayesian Index Policies for Sequential Resource Allocation." *The Annals of Statistics*, **46**(2), 842–865.

Kelly F (1981). "Multi-armed Bandits with Discount Factor Near One: the Bernoulli Case." *Annals of Statistics*, **9**(5), 987–1001.

Steck R (1964). "A Dynamic Programming Strategy for the Two Machine Problem." *Mathematics of Computation*, **18**(86), 285–291.

Villar SS (2018). "Bandit Strategies Evaluated in the Context of Clinical Trials in Rare Life-Threatening Diseases." *Probability in the Engineering and Informational Sciences*, **32**, 229–245.

Williamson SF, Jacko P, Villar SS, Jaki T (2017). "A Bayesian Adaptive Design for Clinical Trials in Rare Diseases." *Computational Statistics and Data Analysis*, **113C**, 136–153.

Yakowitz SJ (1969). *Mathematics of Adaptive Control Processes.* New York, NY: North-Holland.

A. Code

In this section we provide the code of the functions described in this paper.

```
const BB_numerical_precision_64 = 1.0e-13 # instead of 16
const BB_numerical_precision_32 = 1.0e-4 # instead of 7

function DP_2_policy_lin_with_finale( number_of_allocations :: Int64 , value_to_go :: Arra
  # This function implements the DP design for 2 arms
  # value_to_go[1+ number_of_successes_arm_1 ,1+ number_of_failures_arm_1 ,1+ number_of_su
  # Output is the policy (i.e., actions for all states) with linear indices and the Bayes-

        # backwards recursion: the finale (i.e., number_of_observed_responses = number_of_

        # backwards recursion: t-th step
        action_lin = zeros( Int8 , div( number_of_allocations * ( number_of_allocations +
        lin_index = 0
        for number_of_observed_responses = ( number_of_allocations - 1 ) : -1 : 0 , number
            @inbounds begin

                lin_index += 1
                belief_of_success_arm_1 = ( ( prior_success_arm_1 + number_of_successes_ar
                belief_of_success_arm_2 = ( ( prior_success_arm_2 + number_of_successes_ar

                value_to_go_if_action_1 = belief_of_success_arm_1 * ( 1.0 + value_to_go[1+
                value_to_go_if_action_2 = belief_of_success_arm_2 * ( 1.0 + value_to_go[1+

                if ( value_to_go_if_action_1 - value_to_go_if_action_2 ) > BB_numerical_pr

                    value_to_go[1+ number_of_successes_arm_1 ,1+ number_of_failures_arm_1
                    action_lin[ lin_index ] = 1 # action 1 (i.e., arm 1)

                elseif ( value_to_go_if_action_2 - value_to_go_if_action_1 ) > BB_numerica

                    value_to_go[1+ number_of_successes_arm_1 ,1+ number_of_failures_arm_1
                    action_lin[ lin_index ] = 2 # action 2 (i.e., arm 2)

                else #if value_to_go_if_action_1 approx== value_to_go_if_action_2

                    value_to_go[1+ number_of_successes_arm_1 ,1+ number_of_failures_arm_1
                    action_lin[ lin_index ] = 3 # randomise between actions 1 and 2

                end

            end # @inbounds
        end

        return action_lin , value_to_go[1 ,1 ,1 ]

end
```

```
function DP_2_policy_lin( number_of_allocations :: Int64 , prior_success_arm_1 :: Int64 =
  # This function implements the DP design for 2 arms
  # Output is the policy (i.e., actions for all states) with linear indices and the Bayes-

        value_to_go :: Array{ Float64 , 3 } = zeros( Float64 , number_of_allocations + 1 ,
        DP_2_policy_lin_with_finale( number_of_allocations , value_to_go , prior_success_a

end

# Float32 version of value_to_go
function DP_2_action_lin_with_finale( number_of_allocations :: Int64 , value_to_go :: Arra
  # This function implements the DP design for 2 arms
  # Uses linear indexing for value_to_go
  # Output is the immediate action and the Bayes-expected number of successes

        # backwards recursion: the finale (i.e., number_of_observed_responses = number_of_

        # backwards recursion: t-th step
        value_to_go_if_action_1 = 0.0 # needed after the for loop
        value_to_go_if_action_2 = 0.0 # needed after the for loop
        for number_of_observed_responses = ( number_of_allocations - 1 ) : -1 : 0
            value_to_go_lin_index = 0
            for number_of_successes_arm_2 = 0 : number_of_observed_responses , number_of_f
            @inbounds begin

                value_to_go_lin_index += 1
                belief_of_success_arm_1 = ( ( prior_success_arm_1 + number_of_successes_ar
                belief_of_success_arm_2 = ( ( prior_success_arm_2 + number_of_successes_ar

                value_to_go_if_action_1 = belief_of_success_arm_1 * ( 1.0 + value_to_go[ v
                value_to_go_if_action_2 = belief_of_success_arm_2 * ( 1.0 + value_to_go[ v

                if ( value_to_go_if_action_1 - value_to_go_if_action_2 ) > BB_numerical_pr

                    value_to_go[ value_to_go_lin_index ] = Float32( value_to_go_if_action_

                elseif ( value_to_go_if_action_2 - value_to_go_if_action_1 ) > BB_numerica

                    value_to_go[ value_to_go_lin_index ] = Float32( value_to_go_if_action_

                else #if value_to_go_if_action_1 approx== value_to_go_if_action_2

                    value_to_go[ value_to_go_lin_index ] = Float32( ( value_to_go_if_actio

                end

            end # @inbounds
            end
        end
        if ( value_to_go_if_action_1 - value_to_go_if_action_2 ) > BB_numerical_precision_

            action = Int8( 1 ) # action 1 (i.e., arm 1)

        elseif ( value_to_go_if_action_2 - value_to_go_if_action_1 ) > BB_numerical_precis

            action = Int8( 2 ) # action 2 (i.e., arm 2)

        else #if value_to_go_if_action_1 approx== value_to_go_if_action_2

            action = Int8( 3 ) # randomise between actions 1 and 2

        end

        return action , Float64( value_to_go[ 1 ] )

end

function DP_2_action_lin( number_of_allocations :: Int64 , float_version :: Int64 = Int64(
  # This function implements the DP design for 2 arms
  # Uses linear indexing for value_to_go
  # Output is the immediate action and the Bayes-expected number of successes (as Float64

        if float_version == 16

            value_to_go_16 :: Array{ Float16 , 1 } = zeros( Float16 , div( ( number_of_all
            DP_2_action_lin_with_finale( number_of_allocations , value_to_go_16 , prior_su

        elseif float_version == 32

            value_to_go_32 :: Array{ Float32 , 1 } = zeros( Float32 , div( ( number_of_all
            DP_2_action_lin_with_finale( number_of_allocations , value_to_go_32 , prior_su

        else

            value_to_go :: Array{ Float64 , 1 } = zeros( Float64 , div( ( number_of_alloca
            DP_2_action_lin_with_finale( number_of_allocations , value_to_go , prior_succe

        end

end
```

Affiliation:

Peter Jacko
Department of Management Science
Lancaster University Management School
Lancaster, LA1 4YX, UK
E-mail: `p.jacko@lancaster.ac.uk`
URL: `http://www.lancaster.ac.uk/staff/jacko/`