COMP767 - Assignment 1

Alexis Tremblay (260922703)* and Joseph D. Viviano (260878678)*

*University of Montreal

January 2019, 29th

1 Bandit algorithms: Part A [50 points]

1.1 Summary

This paper summarizes all of the main approaches for identifying the arm with the highest mean in a multi-armed bandit problem, attempting to use the smallest number of arm samples as possible, in the *fixed confidence case*. Crucially these algorithms must be able, with a single input δ, to find the arm with the greatest mean with probability of at least $1 - \delta$. A major challenge in this for this problem is that solutions must dynamically adjust sampling to deal with the case that two arms have arbitrarily close means.

The paper gives a summary of the two classes of algorithms developed to address this problem: *action elimination* algorithms, which successively eliminate arms after satisfying a criteria, and *upper confidence bound (UCB)* algorithms which treat the mean of all samples from each arm as the best possible scenario, leading the algorithm to preferentially sample the arm that has provided the best reward in the past (i.e., this algorithm is optimistic). These two algorithms, or variations thereof, dominate this problem space.

In general, the action elimination algorithm decides when to eliminate an arm in question when the mean return of that arm is more than some confidence value lower than the best performing arm. This confidence value can be calculated in a few ways, these are detailed in the paper and they all work in practice, there are only proofs shown for some. Crucially, this confidence value shrinks over iterations to ensure convergence, the rate of which is controlled by the input δ. In contrast, the UCB algorithm samples optimistically after obtaining a single measurement from each arm. This, in effect, allows the algorithm to minimize regret even if it ends up deciding to sample from the non-optimal arm for many iterations, because it can safely ignore the arms who have low upper bounds. They also introduce a variant of UCB (called lower upper confidence

bound, or LUCB) which behaves similarly, but samples the top two arms each iteration. This means that LUCB effectively explores more than the UCB variant.

The paper proceeds to prove the sample complexities of these algorithms, contributing the use of the law of iterated logarithm. The proofs show in general that the algorithms all terminate with the best arm chosen and that the number of arm samples required for convergence can be predicted from the problem parameters.

The remainder of the paper evaluates the average behaviour of the algorithms empirically. We reproduced these main results below. In summary, it is shown that action elimination has the same sample complexity as UCB and LUCB, only in the case that one wants to determine the best arm with at least $1 - \theta$ probability. In practice, when one is willing to accept more uncertainty, these proofs hide some aspects of the algorithms' performances. The action elimination strategy gives little indication of the best arm until after many pulls. On the other hand, both UCB and LUCB give a strong indication of the best arm early on, even if they require many more iterations to terminate.

1.2 Result Replication

For replication, the following parameters were used: 6 arms with the means 1.0, 0.8, 0.6, 0.4, 0.2, and 0, all having variances of 0.25, were used. Each algorithm was repeated 1000 times, and the mean of all repeats are reported. For calculating the C parameter, $\epsilon = 0.01$, $\delta = 0.1$, and $= 1$ (for UCB). For the average reward plot shown, if multiple arms were pulled during an iteration, we stored the mean of those pulls as the reward for that iteration.

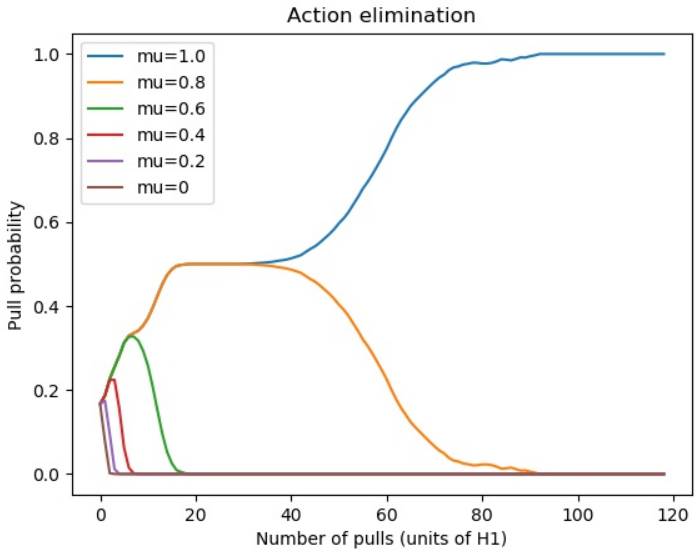

Figure 1: Action Elimination Results

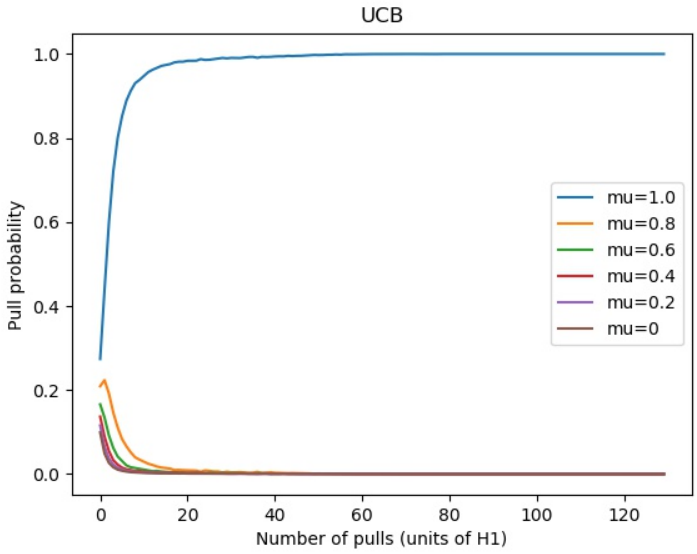

Figure 2: UCB Results

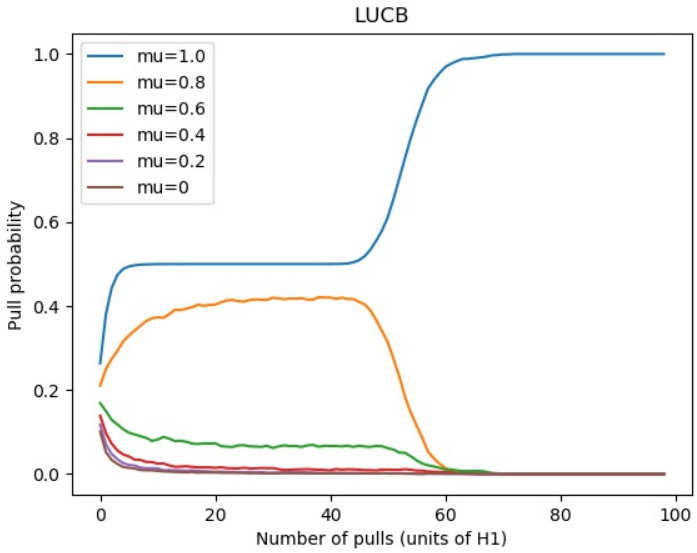

Figure 3: LUCB Results

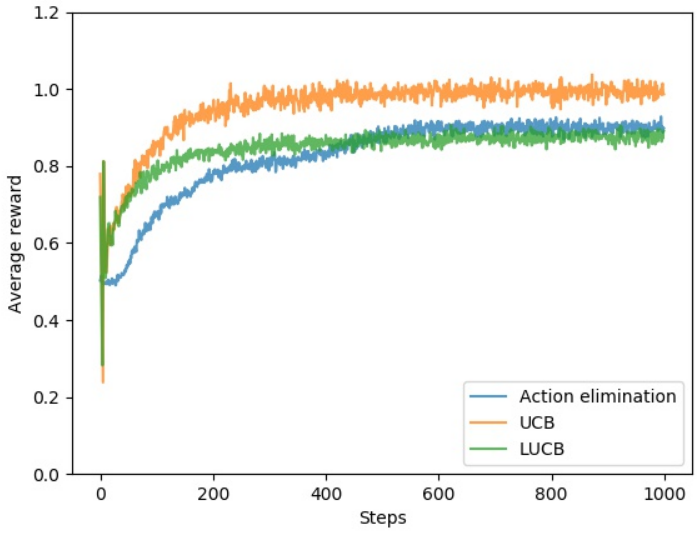

Figure 4: Average reward for the first 1000 steps for all 3 algorithms.

1.3 Regret Minimization vs Best Arm Identification

Since regret minimization can quickly identify a good (if not optimal) solution much earlier than best-arm identification approaches can, regret minimization seems like the better approach in situations where exploration is extremely costly and undesirable, or in situations where the optimal decision may change over time. For example, an algorithm designed to allocate capital into (diversified) investment vehicles would do well to allocate a lot of the capital into a few investment vehicles that show early signs of good performance, instead of continually purchasing under-performing or losing investment vehicles to ensure that, eventually, the algorithm identifies the absolute best investment vehicle. There are two reasons for this. First, the investors caught up in this exploration would be extremely upset with the unnecessary losses. Second, by the time the algorithm converges to the best investment vehicle, there's no telling if it will stay that way forever. If the best arm is expected to change occasionally over time, rapid regret optimization would produce much better rewards.

2 Markov Decision Processes and dynamic programming: Part A [50 points]

2.1 Implementation Explanation

We coded the policy evaluation in matrix form. For this we needed to build a transition matrix that compiles the probability of going from state S to state S'. As an intermediary step we used a 3D matrix with a depth equal to the number of actions. Every layer acts as a transition matrix with respect to every actions by also considering the action probability coming from the policy. Summing over the depth then gives us the full transition matrix irregardless of the action. The 2D transition matrix helped during the evaluation and the 3D transition matrix helped during the policy improvement to get the $argmax$ over the actions.

The following graphics show the value function of the bottom left and right corner during the evaluation of the policy for each policy update.

2.2 Policy Iteration

For the small grid of 5x5 (figure 5 and 6), there is only two policy updates during the training. The first curve (in orange) is the value when the policy is simply random between all actions. The rewards from the top left and right corners does not yet propagate very well to other states due to the complete randomness of the actions.

The value function after an update (blue curves) is higher after updating the policy. The rewards have an easier time to irradiate to all other states. Then we can observe that the value of the bottom right corner is higher than the bottom left. This follows the intuition that the bottom right corner has a lower

Manhattan distance to the top right corner, where the reward is highest, than the bottom left corner. The same logic applies to all other graphics below, so we won't go over this again.

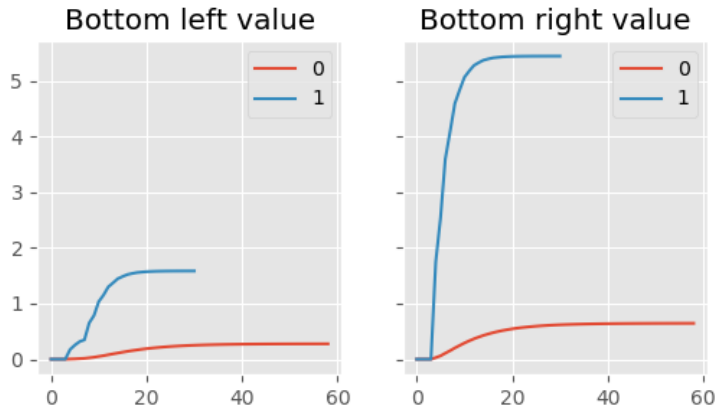

Figure 5: Policy Iteration, p=0.7, n=5

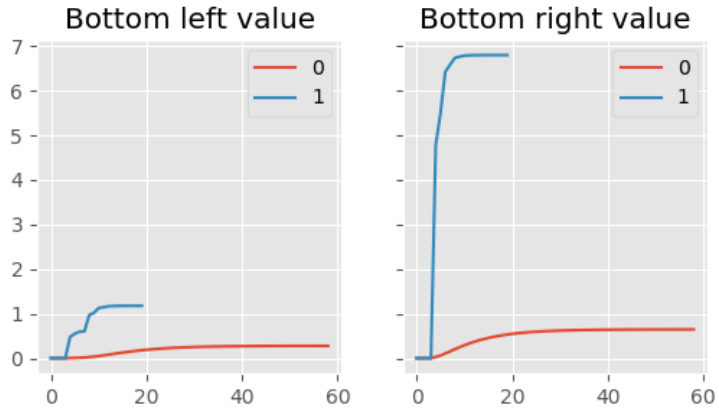

Figure 6: Policy Iteration, p=0.9, n=5

Here is the optimal policy for probability 0.7 and a grid size of 5. We see the absorbing/terminal states in the top corners.

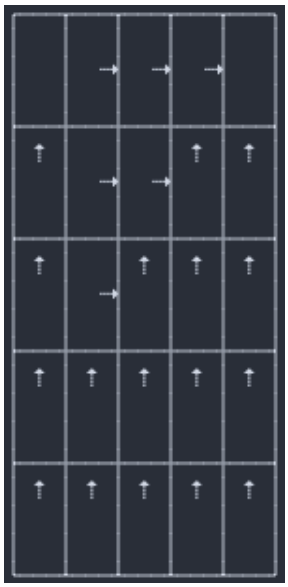

Figure 7: Optimal Policy, p=0.7, n=5, discount=0.9

As a complement of information we show what the optimal policy would look like with a discount of 0.99 instead of 0.9. The arrows now point more to the right. That high reward terminal state act as a more powerful magnet, like bugs on a spotlight at night during a baseball game.

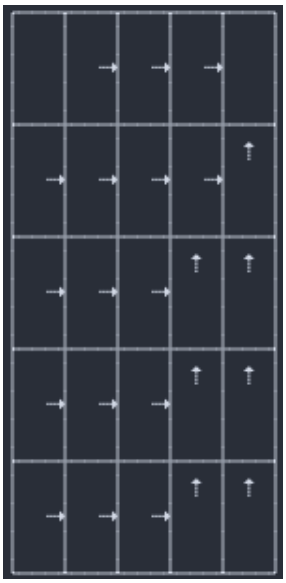

Figure 8: Optimal Policy, p=0.7, n=5, discount=0.99

Figure 9 and 10 are Policy Iteration with the big grid of 50x50. This one takes three updates before finishing. As we can judge by the number of iterations (x axis), it took more time to converge. Since the grid is much bigger than the previous one, it takes more iteration during the evaluation for the rewards to fully propagate to all states.

It's only on the third update that the bottom left and right states start having values different than zero.

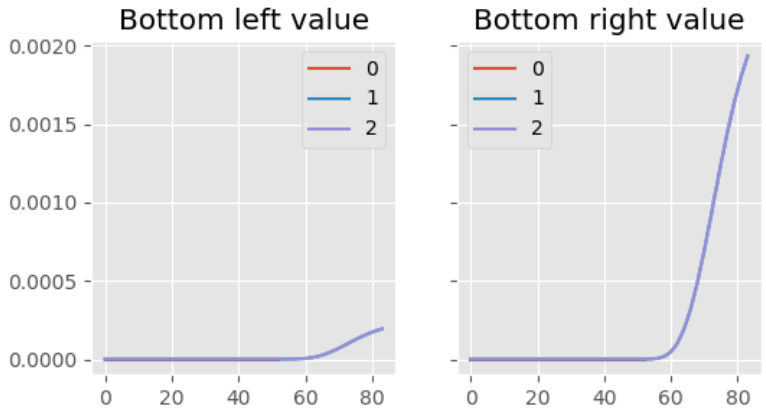

Figure 9: Policy Iteration, p=0.7, n=50

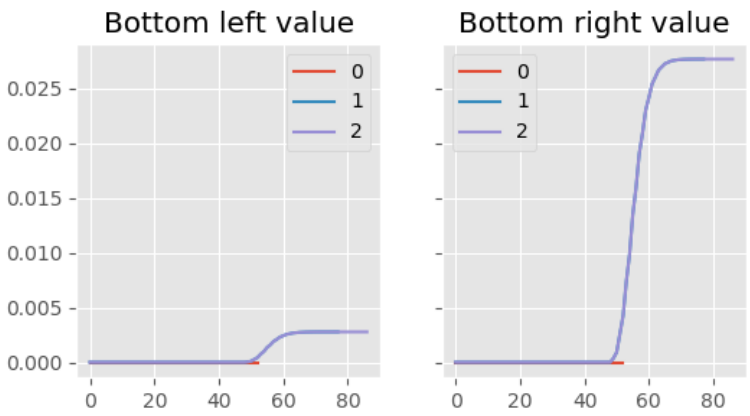

Figure 10: Policy Iteration, p=0.9, n=50

2.3 Value Iteration

For the Value iteration there is only one update, hence the unique curve on the figures 11, 12, 13 and 14. The same logic as Policy iteration applies here; the values in the bottom left and right corners are lower for $p = 0.7$ than for $p = 0.9$ because of the randomness in the actions. The latter is more sure-footed than the former, so its value is higher. It also stops pretty quickly due to the nature of the algorithm. Since Value Iteration gives the same convergence guarantee as Policy Iteration, it makes this algorithm quite interesting to use.

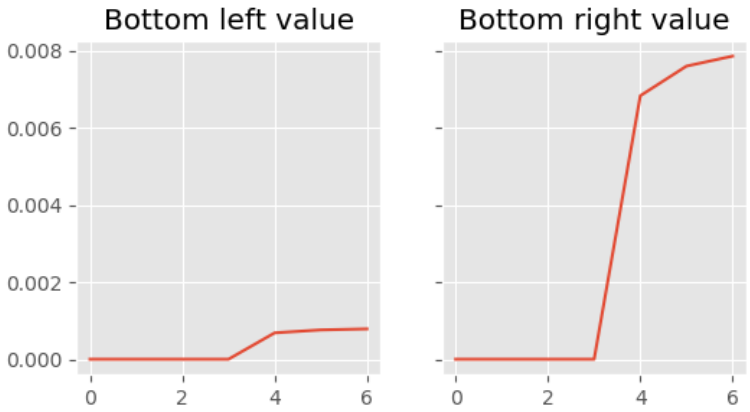

Figure 11: Value Iteration, p=0.7, n=5

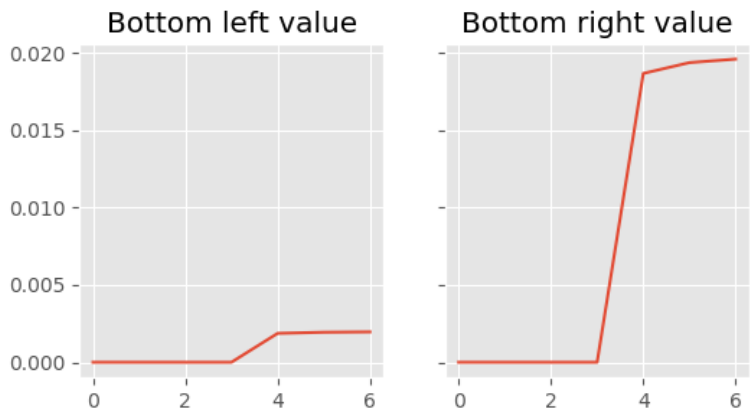

Figure 12: Value Iteration, p=0.9, n=5

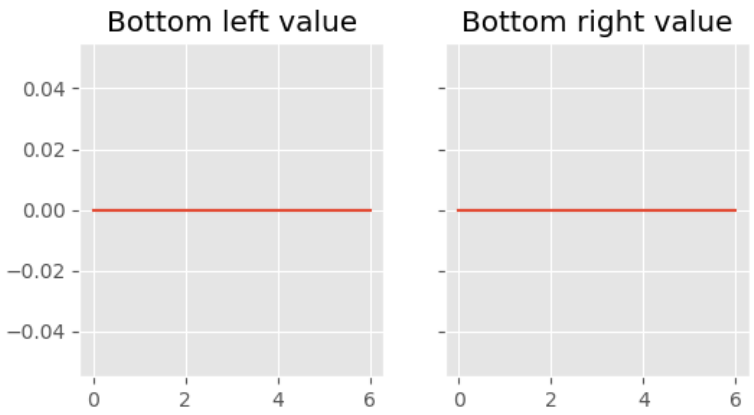

Figure 13: Value Iteration, p=0.7, n=50

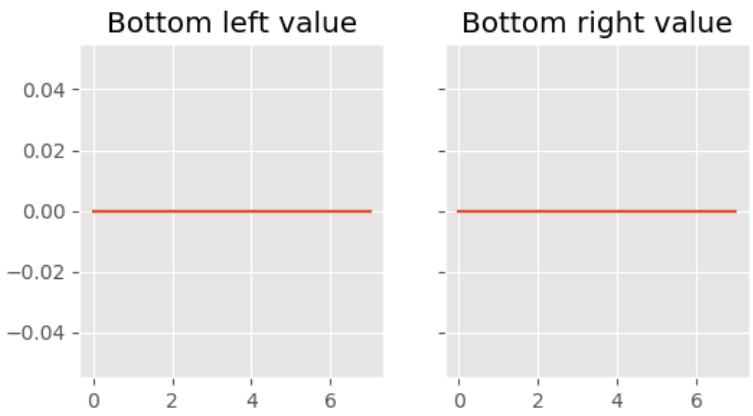

Figure 14: Value Iteration, p=0.9, n=50

2.4 Modified Policy Iteration

With the modified policy the evaluation of the state-value function is only done 10 times (k is a hyperparameter). So it's only normal that the curves fit the early iterations of policy iteration. The state values are a little bit lower than policy iteration, which again makes sense since we are not iteration through the evaluation as many times.

In this case, for $n = 5$ and $n = 50$, both policy and modified policy iteration needs the same number of policy updates before converging. This environment with only two rewards and terminal states makes it fast to converge to the optimal policy, that is why we do not see much difference. We suspect that with a much more difficult environment both iterations would converge at a different rate.

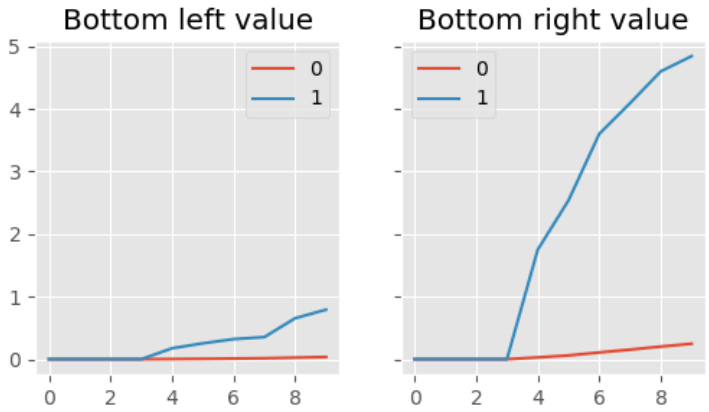

Figure 15: Modified Iteration, p=0.7, n=5, k=10

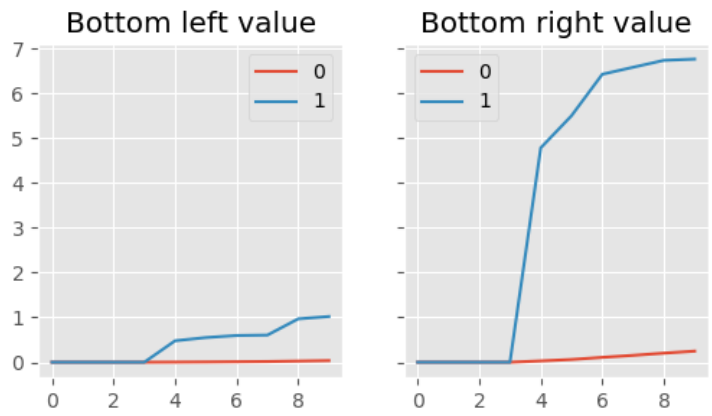

Figure 16: Modified Iteration, p=0.9, n=5, k=10

The following two figures are for $n = 50$. Unsurprisingly the values stay at zero all along. The evaluation is limited to 10 iterations, so there isn't enough time for those states to update their value to something other than zero. If the policy hadn't converge as fast then we might have seen higher values.

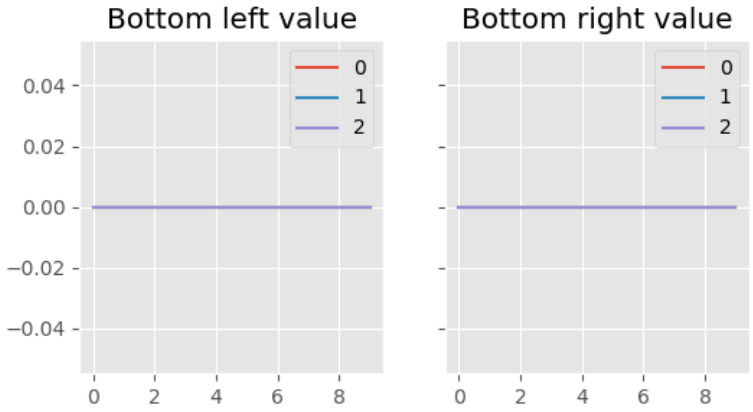

Figure 17: Modified Iteration, p=0.7, n=50, k=10

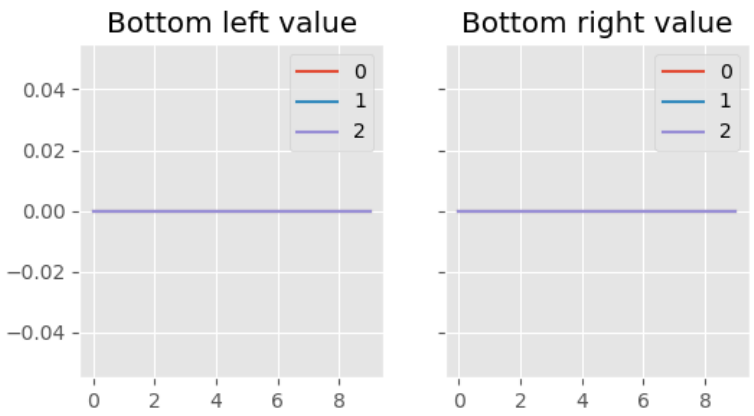

Figure 18: Modified Iteration, p=0.9, n=50, k=10