Replication Package for

The (Surprising) Sample Optimality of Greedy Procedures for Large-Scale Ranking and Selection

by

Zaile Li, Weiwei Fan, and L. Jeff Hong

The paper tests the performance of a class of greedy procedures (including the greedy procedure, the EFG procedure, the EFG$^+$ procedure, and the EFG^{++} procedure) in solving several large-scale ranking and selection problems. Almost all the codes are written in Python and all the experiments are conducted in Jupyter Notebook. No dataset is used in the experiments. If any problems encountered, please contact zaileli21@m.fudan.edu.cn.

1. General Introduction

The replication package includes code files and Jupyter notebooks. They can be used on both Mac OS and Windows systems. **To produce the Figures and Tables in the paper, just run the Jupyter notebooks. Python 3.8 or 3.9 is required** and the following libraries are used:

- *numpy, scipy, copy*: for basic computing functionalities and random number generations
- *matplotlib*: for producing all the figures.
- *ray*: for parallelization of the multiple-replication experiments.
- *multiprocessing:* for a parallel implementation of the parallelizable EFG^{++} procedure.
- *time, win_precise_time (only for Windows)*: for time logging and time control in the simulation studies. For the experiment with random simulation times described in EC 4.3.2 of the paper, we use the library win_precise_time to control the sleep of the parallel processors, as shown in the function TpMax_pause (utils.py). We do this because, on Windows, the time.sleep() function has an unsatisfying precision level. This is not an issue on MAC OS systems. Check https://pypi.org/project/win-precise-time/ for the lib.
- *tqdm*: for progress bar (rarely used).
- *Jupyter Notebook*: for conducting the experiments and analyze the results.
- *Cython*: for compiling the simulation program of the practical throughput maximization problem used in Section 6.4 and EC 4 into C lib to accelerate the simulation.

The codes are *insensitive* to the version of Python as well as the libraries. We recommend the reader to use Anaconda to install and manage these libraries. In the following, we first list the experiment setting and notebooks in Section 2. We then introduce the code files in Section 3.

2. (MUST-READ) Experiment Setting and Notebook Files

The experiments are mainly conducted to compare the probability of correct selection (PCS) or probability of good selection (PGS) of different procedures (i.e., algorithms). When solving a particular problem with a problem scale parameter *k*, the correct (good) selection event means that, the procedure successfully identifies and returns the index of the true best (one good) alternative among a finite set of alternatives. To estimate the average performance of a procedure, i.e., the probability of success, it is necessary to run the procedure multiple (1000 or 2000) times for even one single problem (one value of *k*). Furthermore, in the paper, the figures typically display how the estimated PCS change as *k* increases and thus the PCS for multiple values of *k* need to be estimated. Consequently, for producing even one figure, finishing the experiments using a single computing thread may take several hours. To accelerate the experiments, we use a convenient Python library Ray to distribute the multiple replications to multiple computing threads.

The experiments included are all conducted on lab servers with more than 50 CPU cores (supporting more than 100 concurrent computing threads) and 512 GB memory. In each experiment notebook, the Ray library is used first to open 96 parallel computing threads for the following computing tasks. The codes for Ray initialization are show as follows.

```
num_cpus=96
ray.shutdown()
ray.init(num_cpus=num_cpus, ignore_reinit_error=True)
```

Even if your PC or server does not have 48 CPUs to support 96 parallel threads, you CAN also run the codes. But, when running the computing blocks, the following warning may appear. You can ignore the warnings and wait for the completed results. The total computing time depends on how many CPUs are TRULY used and the performance of your CPUs. It may take HOURS to finish one single experiment if you only have 8 CPUs on your computer. A 512 GB of memory is NOT necessary, but there is a risk of memory overload if it is too small.

```
WARNING: Python worker processes have been started on node: with address: 127.0.0.1. This could be a result of using a large number of actors, or due to tasks blocked in ray.get() calls (see https://github.com/ray-project/ray/issues/3644 for some discussion of workarounds).
```

You may change the num_cpus to a smaller number to avoid the warnings or overload of yo ur computer.

The following is a full list of notebooks used to produce the Figures and Tables. Please check the files to run the experiments:

Experiments in the Main Paper	Experiments in the Online Companion
➢ Figure_1.ipynb *Figure 2 is not based on numerical results. It is only an illustrative example.* ➢ Figure_3_Greedy_Procedure.ipynb	➢ Figure_EC3.ipynb ➢ Figure_EC4.ipynb ➢ Figure_EC5.ipynb ➢ Figure_EC6.ipynb

➢ Figure_3.ipynb ➢ Figure_4_EC1.ipynb ➢ Figure_5_EC2.ipynb ➢ Table_2and3_Part1.ipynb ➢ Table_2and3_Part2.ipynb ➢ Table_2and3_SH.ipynb	➢ Figure_EC7_8.ipynb ➢ Figure_EC9.ipynb ➢ Figure_EC9_PCS.ipynb ➢ Figure_EC10_11.ipynb ➢ Figure_EC12_13.ipynb ➢ Figure_EC14.ipynb ➢ Table_EC12344.ipynb ➢ Table_EC5.ipynb

3. Code Files

File Name	Introduction
procedures.py includes all compared procedures: ✧ *EFG* ✧ *Greedy (set n_0=1 in EFG)* ✧ *EA (set n_g=0 in EFG)* ✧ *EFGPlus* and the compared ones: ✧ *SH* ✧ *Modified_SH* ✧ *FBKT (seeding=False)* ✧ *FBKT-Seeding (seeding=True)* ✧ *OCBA* **procedures_commented.py** includes the comments for the compared procedures	The procedures are developed following the same template. They receive a generator (which is an instance of a specific Generator class) and other procedure parameters. The generator itself represents an R&S problem instance, keeps track of true values of alternatives in the problem, and generates simulation observations of each alternative. When the procedures exhaust the user-specified sampling budget, they return the ID of the alternative that is selected as the best. When a procedure is used to solve a particular problem instance, it is executed multiple times to estimate the mean performance, i.e., the probability of correct selection (PCS) or the probability of good selection (PGS). To speed up, we use the Python library Ray to parallelize the repeated procedure runs on multiple CPUs (the procedures themself are not parallelized). Each procedure has two versions: the vanilla version and the remote_version. The vanilla version is mainly used for validation/debugging while the remote version (decorated by @ray.remote) is used by Ray in the parallel experiments.
utils.py includes all the common functions	When solving a problem instance, the procedure will be executed many (e.g., 1000) times to estimate its

required for conducting the experiments and analyzing the results: *1. parallel_experiments* *2. evaluate_PCS* *3. evaluate_PGS* *4. SCCVGenerator* *5. EMCVGenerator* *6. EMIVGenerator* *7. EMDVGenerator* *8. loadTPMax* *9. TpMax* *10. TpMaxGenerator* *11. TpMax_pause* *12. TpMaxGeneratorPause* *13. process_result* *14. evaluate_PCS_parallel* 15. *evaluate_PGS_parallel* **utils_updated.py** The TpMaxGenerator function in utils.py is slightly changed during the review process (a random shuffle step of the alternatives is removed). The change does not impact the main conclusions. To produce exactly the same results, we create a utils_updated.py.	PCS or PGS. These replications are finished in parallel based on Ray. parallel_experiments receives the generators and the remote version of the procedure, finishes the repeated procedure runs using Ray, collects the results, and estimates the PCS and PGS (based on evaluate_PCS and evaluate_PGS). Generators are the class of R&S problems. The instantiated generators keep track of the true values of alternatives in the problem instance and generate simulation observations of each alternative. TpMax functions are used for testing the procedures on the throughput maximization (TP) problem. loadTPMax loads the true mean values of the alternatives from a CSV file for a particular problem instance (see the introduction of solve_means.m). TpMax is the simulator of each alternative and TpMaxGenerator is the Generator for the TP problem. TpMax pause functions are the modified version with explicit control of the simulation time. See EC.4.3.2 of the paper for an introduction. process_result analyzes the results of the experiments in EC 4 of the paper based on evaluate_PCS_parallel and evaluate_PGS_parallel to produce Table EC1-5 for the parallel EFG^{++} procedure (in the file EFGPlusPlus.py).
solve_means.m **lpsolvefun.m** These files are MATLAB codes for solving the true means of each alternative in the throughput maximization (TP) problem. See EC.3.4.2 for an introduction. **results_20_20.csv** **results_30_30.csv** **results_45_30.csv** **results_45_45.csv**	The TP problem has two parameters S1, S2, e.g., 20, 20. They decide the problem scale. The function solve_means requires the two parameters and utilizes lpsolvefun.m to solve the underlying balancing equations of the underlying Markov chains (See EC 3.4.2 of the paper). The .csv files include the real mean values of all alternatives for the problems included in Table 1. To load the values and associate them with their respective alternatives, see loadTPMax (in the file utils.py).

TpMaxSimulator.pyx **setup.py** TpMaxSimulator.c TpMaxSimulator.obj TpMaxSimulator.cPython-38-darwin.so TpMaxSimulator.cPython-39-darwin.so TpMaxSimulator.cPython-310-darwin.so TpMaxSimulator.cp38-win_amd64.pyd TpMaxSimulator.cp38-win_amd64.lib TpMaxSimulator.cp38-win_amd64.exp TpMaxSimulator.cp39-win_amd64.pyd TpMaxSimulator.cp39-win_amd64.lib TpMaxSimulator.cp39-win_amd64.exp	The .pyx file includes the simulation logic codes of the throughput maximization problem used in Section 6.4 of the paper. It is called by the TpMax functions. The codes are translated from a Java version (at https://github.com/biazhong/FBKT/blob/778fb1ab999de22ab85ee3d3ae1d3c9f3a1d52fc/FBKTSpark/src/main/java/problems/TpMax.java). To speed up the simulation, we use Cython to compile it into C lib, which is then used as a Python extension module. The setup.py is used for this. See https://cython.readthedocs.io/en/latest/src/quickstart/build.html (Building a Cython module using setup tools) for guidance on how to build the .pyx file into a Python extension module. The .so files are the produced modules for different Python versions on MAC OS. The .pyd, .lib, and .exp files are the produced modules on Windows. If your Python version is not covered, compile the .pyx file. WARNING: Due to MAC OS's security check on external software, the first time you use TpMaxSimulator you need to allow external software permission in Settings - Privacy and Security.
EFGPlusPlus.py includes the implementation of the parallelized EFG^{++} procedure: 1. *EFGPlusPlus* 2. *__EFGPlusPlus* 3. *sequential_filling*	The implementation of the EFG^{++} procedure utilizes the Python library multipleprocessing in a master-worker structure. See EC.4.1 of the paper for the details. The master thread controls a bunch of worker threads. The EFGPlusPlus functions include the codes of the master, while the __EFGPlusPlus function includes the codes of each worker. The sequential_filling function is used to solve the load-balancing simulation task assignment as introduced in EC.4.1 and EC 4.4 of the paper.