# Scaling Exact Inference for Discrete Probabilistic Programs Artifact Overview

Steven Holtzen and Guy Van den Broeck and Todd Millstein
{sholtzen,guyvdb,todd}@cs.ucla.edu

Thank you for taking the time to evaluate this artifact. In this document you fill find detailed instructions on how to build the artifact and how to reproduce the experiments from the original paper.

## 1 Getting Started

### 1.1 Building the Docker Image

The building process in has only been tested on Mac and Linux. To make building easier, we provide a docker image, which is our recommended method for installation. First, follow the instructions at `https://docs.docker.com/get-docker/` to install Docker. Then, to download and run the image, execute the following in your terminal:

```
1  docker pull sholtzen/dice-oopsla2020:1.0
2  docker run -it sholtzen/dice-oopsla2020:1.0
```

This will put you in an interactive virtual machine terminal with `dice` compiled. The compiled binaries can be found by navigating to the following directory in the docker image:

```
1  cd /home/project/dice/
```

In this directory you fill find two binaries:

- `Dice.native`: the main `dice` interpreter. This program expects a single command-line argument, which is the name of a `dice` file to interpret.
- `Run_bench.native`: used to run the benchmarking experiments from the main paper (more details on this later).

### 1.2 Building from Source

We provide a source tarball in the upload. In this tarball, there is a file called `README.md` which contains build instructions for compiling from source, along with the basics on `dice` syntax.

## 2 Evaluating the Artifact

There are two main files in this artifact:

- `Dice.native`, which interprets dice files;
- `Run_bench.native`, which reproduces the experiments from the paper.

Here we will describe how to use each of these files.

## 2.1 Running a Simple Dice Interpreter

This is a good test to "kick the tires" of the interpreter. The following is a simple `dice` program that flips two coins and observes one of them to be heads, and then asks for the posterior probability that one of the coins landed heads:[1]

```
let a = flip 0.3 in
let b = flip 0.8 in
let tmp = observe a || b in
a
```

This file is already included in the source repository, and can be found in `resources/example.dice`. From the Docker VM, we can evaluate the example by executing the following:

```
$ cd /home/project/dice/
$ ./Dice.native resources/example.dice
Value Probability
true 0.348837
false 0.651163
Final compiled size: 5
Live: 12
```

This prints the posterior probability of the coin `a`, along with the final compiled BDD size. It says that `a` is `true` with probability 0.348837. The number of live nodes ("Live: 12") is also printed, for debugging purposes.

## 2.2 Reproducing the `Dice` Performance Experiments

The evaluations from the paper can be readily reproduced by running the `Run_bench.native` file, which is again found in the Docker VM in the following directory: `/home/project/dice`.

To reproduce the `dice` performance results from the paper, we evaluate this program with the following arguments:

- `Run_bench.native -baselines`: reproduces the experiments in Table 1. This produces an output that looks like:

```
************************************[Baselines]************************************
Benchmark Time (s) #Paths (log10) BDD Size
benchmarks/bayesian-networks/full-joint/hailfinder.dice 1.248486 76.2644830539 213745
benchmarks/bayesian-networks/full-joint/alarm.dice 0.618520 36.0163161601 437658
benchmarks/bayesian-networks/full-joint/hepar2.dice 0.159968 69.4591645096 54860
benchmarks/bayesian-networks/full-joint/insurance.dice 0.477508 40.9288602533 232111
...
benchmarks/bayesian-networks/water.bif.dice 41.749295 54.5040846322 51952
benchmarks/bayesian-networks/cancer.bif.dice 0.006813 3.06145247909 28
benchmarks/bayesian-networks/pigs.bif.dice 2.603573 492.866256125 35
benchmarks/bayesian-networks/insurance.bif.dice 0.233304 41.0965509703 101047
benchmarks/bayesian-networks/munin.bif.dice 4.247231 1622.31896525 11977
benchmarks/bayesian-networks/hailfinder.bif.dice 0.809843 76.3058757391 65386
```

[1] See the `README.md` file of the attached source tarball for detailed documentation on the syntax of `dice`

```
benchmarks/bayesian-networks/hepar2.bif.dice 0.031008 69.4591645096 3936
benchmarks/bayesian-networks/alarm.bif.dice 0.017109 36.0163161601 1366
benchmarks/bayesian-networks/survey.bif.dice 0.002633 4.14063372513 73
benchmarks/baselines/twocoins.dice 0.002214 0.602059991328 5
benchmarks/baselines/alarm.dice 0.001152 1.98227123304 11
benchmarks/baselines/evidence1.dice 0.002157 0.903089986992 5
benchmarks/baselines/digitRecognition.dice 0.341204 237.706486605 7896
benchmarks/baselines/noisyOr.dice 0.001202 4.2144199393 35
benchmarks/baselines/murderMystery.dice 0.001133 1.20411998266 6
benchmarks/baselines/evidence2.dice 0.001016 0.903089986992 6
benchmarks/baselines/grass.dice 0.001149 2.40823996531 15
benchmarks/simpleCaesar.dice 0.632390 342.423550209 84968
```

This output is a tab-separated table. The order in which the rows are printed is non-deterministic, and can take several minutes to run. In the VM environment, these results will be slower than the original paper reports. The results from Table 2 will be listed as "full-joint".

- `Run_bench -caesar`: reproduces the Caesar-cipher experiments in Figure 10(a). It generates the following table:

```
opam@bd248bf2f4ed:/home/project/dice$ ./Run_bench.native -caesar
**************************************[Caesar No Inline]**************************************
Length Time (s) BDD Size
1 38.419962 2.82994669594 377
100 197.171926 142.912308145 35126
250 569.227934 355.158310341 87776
500 1141.479015 708.901647333 175526
1000 2309.463978 1416.38832132 351026
2500 5703.559875 3538.84834327 877526
...
```

This experiment will likely take several hours to run (it goes to 50000). First, it evaluates the inlined performance, then it evaluates the non-inlined performance.

- `Run_bench -diamond`: reproduces the diamond network experiments in Figure 10(b). It generates the following table:

```
./Run_bench.native -diamond
**************************************[Diamond Non-Inlined]**************************************
Length Time (ms) BDD Size
1 4.754066 2.40823996531 6
100 8.570194 121.616118248 204
200 26.407003 242.028116514 404
300 16.404152 362.440114779 604
400 11.821985 482.852113045 804
500 14.523983 603.264111311 1004
700 12.488127 844.088107842 1404
800 15.290022 964.500106107 1604
900 31.028986 1084.91210437 1804
1000 19.703865 1205.32410264 2004
2000 57.881117 2409.44408529 4004
3000 103.801966 3613.56406795 6004
4000 119.899035 4817.68405061 8004
5000 147.032022 6021.80403326 10004
**************************************[Diamond Inlined]**************************************
Length Time (ms) BDD Size
1 1.718998 2.40823996531 6
100 9.840965 121.616118248 204
200 9.957075 242.028116514 404
300 28.625011 362.440114779 604
400 55.752039 482.852113045 804
500 44.849873 603.264111311 1004
700 54.716110 844.088107842 1404
800 53.364038 964.500106107 1604
900 63.953876 1084.91210437 1804
1000 79.336166 1205.32410264 2004
2000 124.580860 2409.44408529 4004
3000 181.603909 3613.56406795 6004
4000 208.308935 4817.68405061 8004
5000 275.877953 6021.80403326 10004
```

- `Run_bench -ladder`: reproduces the ladder network experiments in Figure 10(c). It generates the following table:

```
./Run_bench.native -ladder
************************************[Ladder Inlined]************************************
Length Time (ms) BDD Size
1 5.263090 1.90848501888 12
100 11.246920 96.3784934534 606
200 16.692877 191.802744397 1206
300 32.870054 287.226995341 1806
400 33.671856 382.651246285 2406
500 62.623024 478.075497229 3006
700 59.226036 668.923999117 4206
800 77.877045 764.348250061 4806
900 69.668055 859.772501005 5406
1000 103.832960 955.196751949 6006
2000 146.524191 1909.43926139 12006
3000 203.852892 2863.68177083 18006
4000 261.821985 3817.92428027 24006
5000 295.618057 4772.16678971 30006
****************************************[Ladder Non-Inlined]****************************************
Length Time (ms) BDD Size
1 4.740953 1.90848501888 12
100 4.276991 96.3784934534 606
200 12.408972 191.802744397 1206
300 8.482933 287.226995341 1806
400 10.219812 382.651246285 2406
500 15.550852 478.075497229 3006
700 26.414871 668.923999117 4206
800 30.678034 764.348250061 4806
900 54.786921 859.772501005 5406
1000 50.775051 955.196751949 6006
2000 132.426023 1909.43926139 12006
3000 185.606003 2863.68177083 18006
4000 230.031967 3817.92428027 24006
5000 312.528849 4772.16678971 30006
```

- `Run_bench -motivating`: reproduces the motivating experiments in Figure 10(d). It generates the following table:

```
./Run_bench.native -motivating
************************************[Motivating]************************************
Length Time (ms) BDD Size
1 4.655123 1.90848501888 12
100 8.017063 96.3784934534 606
200 8.463144 191.802744397 1206
300 23.199081 287.226995341 1806
400 29.350042 382.651246285 2406
500 49.848080 478.075497229 3006
700 51.189899 668.923999117 4206
800 62.526941 764.348250061 4806
900 62.072039 859.772501005 5406
1000 97.001076 955.196751949 6006
2000 139.187098 1909.43926139 12006
3000 189.225912 2863.68177083 18006
4000 229.627848 3817.92428027 24006
5000 275.480032 4772.16678971 30006
```

## 2.3 Running Comparisons

### 2.3.1 Psi

We primarily compare against the Psi probabilistic programming engine. This is included in the provided Docker image. To run the Psi performance benchmarks, a Python tool is provided and can be executed by running the following in the Docker image:

```
python3 /home/run-psi.py
```

### 2.3.2 WebPPL

The WebPPL system is also installed in the included Docker image. For instance, to run the diamond experiments, execute:

```
webppl /home/project/dice/benchmarks/diamond-network.webppl
```

To adjust the lengths for the webppl examples, add or remove calls to the relevant scaling function. For instance, as a clarifying example, the encrypt.webppl file looks like:

```
var encrypt = function(key, observedChar) {
    var sampledChar = sample(Discrete({ps: [0.08167, 0.01492, 0.02782,
                            0.04253, 0.12702, 0.02228, 0.02015, 0.06094,
                            0.06966, 0.0153, 0.0772,
                            0.04025, 0.02406, 0.06749,
                            0.07507, 0.01929, 0.00095, 0.05987,
                            0.06327, 0.09056, 0.02758, 0.00978,
                            0.02360, 0.00150, 0.01974, 0.00074]}));
    var encrypted = (sampledChar + key) % 26;
    condition(encrypted == observedChar)
};


var main = function() {
  var key1 = sample(Discrete({ps: [0.038461538,0.038461538,0.038461538,
                            0.038461538,0.038461538,0.038461538,
                            0.038461538,0.038461538,0.038461538,
                            0.038461538,0.038461538,0.038461538,
                            0.038461538,0.038461538,0.038461538,
                            0.038461538,0.038461538,0.038461538,
                            0.038461538,0.038461538,0.038461538,
                            0.038461538,0.038461538]}));

    encrypt(key1, 0);
    encrypt(key1, 1);
    // .. many more encrypt calls

  return key1;
};

var dist = Infer({method: 'rejection'}, function() {
  return main();
});

display(dist);
```

The number of calls to encrypt is the length. To get different lengths, add or remove calls to encrypt from this file, and rerun it with webppl. A similar pattern holds for the other webppl files.