Disease Control System DiCon

Sebastian Goll, Ned Dimitrov

University of Texas at Austin

November 24, 2009

What is DiCon?

DiCon is the Disease Control System.

Procedure

- Use any disease simulator.
- Define policy/set of parameters.
  (e. g., when or where to distribute antivirals)
- Simulate for different policies and find best one.

Distributed system

- DiCon runs on computer clusters.
- Many simulator instances work in parallel.
- Optimization algorithm picks policies to simulate next.

What is a policy?

Policy describes parameters to optimize.

- Set of parameters that influence simulation outcome.
  (e. g., how many antivirals to distribute at each month)
- Each simulation returns a reward value, between 0 and 1.
- Optimizer finds policy that gives best average reward value.

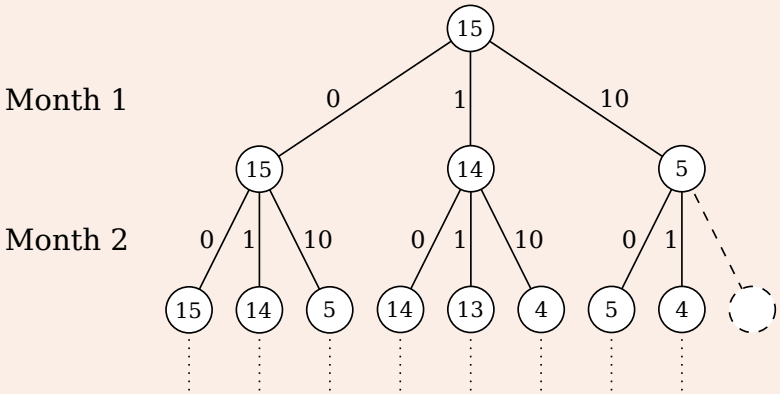
Stochastic simulation

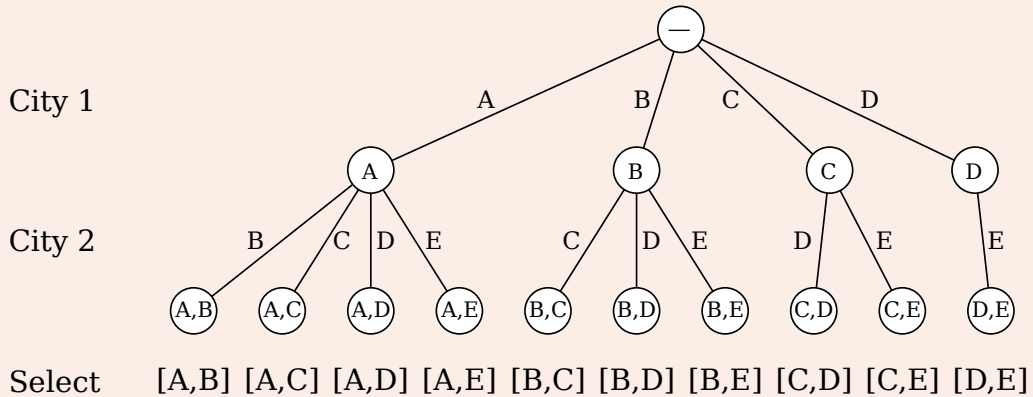
- Simulations can be stochastic.
  (i. e., return different reward value each time they are run)
- Best policy according to average return value over all runs.

Tree structure

- For best performance, policy should have tree structure.
- Tree structure comes naturally for many policies.
- Every list is a candidate for tree structure.

Examples





Design goals

The following goals were followed during development of DiCon.

Ease of use	Easy to manage; single configuration file; intuitive.
Flexibility	Custom simulators & optimizers; simple interface.
Performance	Little overhead; advantage of parallel computing.
Simplicity	Easy to modify when necessary; documentation.

Source code

- $\approx$ 12,000 lines of code (C++).
- Heavy use of Boost C++ Libraries, also Google's Protocol Buffers and GNU MP Bignum Library.
- Works on any cluster with Message Passing Interface (MPI).

Scheme

- Write simulator.
- Define policy space.
- Run DiCon optimizer.

Plaything

- Let k and n be fixed positive integers ($k, n \in \mathbb{N}$).
- Policy is a k-ary decimal number with n digits.
  (e. g., "0.123" with $k = 4$, $n = 3$)
- Policy space is set of numbers between 0 and 1.
  (e. g., "0.000" through "0.333", $k = 4$, $n = 3$)
- Simulator is deterministic; returns number.
- Goal: Find "0.333" ($k = 4$, $n = 3$).

Simulator

Simulator provides both policy space and simulation.

Methods

Simulator defines three simple methods.

- **children** Gets series of policy elements and returns node children.
- **simulate** Gets policy and returns reward value, between 0 and 1.
- **display** Gets policy and returns human-readable interpretation.

Policy: List of policy elements pointing to leaf node in policy tree.

Python code

```
1  from simulator import Simulator
2
3  base = 4  # That's k.
4  depth = 3  # That's n.
5
6  class Plaything( Simulator ):
7      def children( self, path ):
8          if len(path) < depth:
9              return range( 0, base )
10         else:
11             return []
12
13     def simulate( self, policy ):
14         value = 0.0
15         factor = 1.0
16
17         for digit in policy:
18             factor /= base
19             value += digit * factor
20
21         return value
22
23     def display( self, policy ):
24         return str( policy )
25
26 if __name__ == "__main__":
27     Plaything().main()
```

C++ code

```
1  #include "../simulator.hpp"
2  #include <boost/foreach.hpp>
3  #include <boost/lexical_cast.hpp>
4
5  static const size_t base = 4;  // That's k.
6  static const size_t depth = 3;  // That's n.
7
8  class Plaything
9    : public Simulator<int>
10 {
11 public:
12   virtual
13   std::vector<int>
14   children(const std::vector<int> &path) {
15     std::vector<int> res;
16
17     if(path.size() < depth) {
18       for(size_t i = 0; i < base; ++i)
19         res.push_back(i);
20     }
21
22     return res;
23   }
24
25   virtual
26   double
27   simulate(const std::vector<int> &policy) {
28     double value = 0;
29     double factor = 1;
30
31     BOOST_FOREACH(int digit, policy) {
32       factor /= base;
33       value += digit * factor;
34     }
35
36     return value;
37   }
38
39   virtual
40   std::string
41   display(const std::vector<int> &policy) {
42     std::string res;
43
44     BOOST_FOREACH(int digit, policy) {
45       if(!res.empty()) res += ", ";
46       res += boost::lexical_cast<std::string
         >(digit);
47     }
48
49     return '[' + res + ']';
50   }
51 };
52
53 int main() {
54   Plaything().main();
55 }
```

Invocation

- DiCon executable gets name of working directory.
- Directory initially contains only main config file.
- Gets filled with results as DiCon is running.

Example

```
$ mpirun -np 4 ./dicon demo
```

Config file

```
; Sample DiCon configuration file.
;
; Lines starting with ';' are comments.
; The options below show default values.

[global]
;main_logfile=log/main.log
;main_log_level=info
;node_logfile=log/node_%04u.log
;node_log_level=info
;max_jobs=1000000
;job_dir=job-%u
;arg_dir=argument-%05u
;job_logfile=job.log
;job_log_level=info
;checkpoint=checkpoint.xml
;opt_logfile=log/optimizer-node_%04u.log
;sim_logfile=log/simulator-node_%04u.log
;optimizer_map_file=dump/%012u-optmap.bin
;optimizer_lib_file=dump/%012u-optlib.bin
;policy_bin_dumpfile=dump/%012u-policy.bin
;policy_txt_dumpfile=dump/%012u-policy.txt
;policy_count=10
;policy_bin_result=result-policy.bin
;policy_txt_result=result-policy.txt

main_log_level=debug
node_log_level=debug
job_log_level=debug

[job-1]
;max_sims=0
;max_nodes=0
;job_backlog=1
;node_backlog=1
;checkpoint_secs=14400
;checkpoint_sims=0
;checkpoints=
;optimizer=
;simulator=
;opt_args[1]=
;sim_args[1]=

max_nodes=0
job_backlog=1
node_backlog=1
checkpoint_secs=0
checkpoint_sims=0
optimizer=../optimizer/exhaustive.so
simulator=../simulator/python/
        simple_simulator.py

[job-2]
; Some other job set here.
```

Results

When DiCon finishes, working directory might look like this.

```
demo
|-- job-1                                  // (one directory per job section in config file)
|   `-- argument-00001                     // (one sub-directory per argument combination)
|       |-- checkpoint.xml                 // (file describing most recent checkpoint)
|       |-- dump                           // (directory with checkpoint files)
|       |   |-- 000000000025-optlib.bin
|       |   |-- 000000000025-optmap.bin
|       |   |-- 000000000025-policy.bin
|       |   `-- 000000000025-policy.txt
|       |-- job.log                        // (general log file for the current job)
|       |-- log                            // (output from optimizer and simulators)
|       |   |-- optimizer-node_0001.log
|       |   |-- simulator-node_0001.log
|       |   |-- simulator-node_0002.log
|       |   `-- simulator-node_0003.log
|       |-- result-policy.bin              // (best policy found, binary)
|       `-- result-policy.txt              // (best policy, human-readable)
|-- log                                    // (general system-wide log files)
|   |-- main.log
|   |-- node_0001.log
|   |-- node_0002.log
|   `-- node_0003.log
`-- main.conf                              // (initial configuration file)
```

Checkpoints

- DiCon automatically takes checkpoints.
- Contain state dump of each optimizer.
- Can be resumed after interruption.

Useful

- Checkpointing is necessary on some clusters.
- Jobs may be allowed to run only some time (48 hours).
- Resuming is easy: just start DiCon with same directory.
- DiCon automatically recognizes jobs with checkpoints.

Simulator arguments

- Run simulator with command-line arguments.
- Use `sim_args[`n`]` options in config file ($n \geq 1$).
- Special syntax: definition of many jobs at once.

(`sim_args[`n`]` parameter $\longrightarrow$ generated set of values)

```
1  foo|bar                          → {foo, bar}
2  [1..10]                          → {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
3  [0,0.1..1]                       → {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}
4  [1..10|10,20..90]                → {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90}
5  [x^2+y^2:x=1..6:y=2..4]          → {5, 10, 17, 8, 13, 20, 13, 18, 25, 20, 25, 32, 29, 34, 41, 40,
                                      45, 52}
6  --foo=[1..2]|--bar=[3..9]        → {--foo=1, --foo=2, --bar=3, --bar=4, --bar=5, --bar=6, --bar=7,
                                      --bar=8, --bar=9}
7  --method={a|b} --value=[1..2]    → {--method=a --value=1, --method=a --value=2, --method=b --value
                                      =1, --method=b --value=2}
```

(Similar syntax for `checkpoints` parameter)

```
1  1,4,9,2,6,10                     → {1, 2, 4, 6, 9, 10}
2  [2..4|9,7..1]                    → {1, 2, 3, 4, 5, 7, 9}
3  [2^(x*2):0..]                    → {1, 4, 16, 64, 256, ...}
```