I-EPOS Manual

Peter Pilgerstorfer

January 3, 2017

1 Introduction

This documents describes the software that simulates I-EPOS, an algorithm for decentralized combinatorial optimization.

1.1 Installation and setup

- Download the project repository from GitHub (TODO).
- Make sure that a version of Java 8 is installed. You can download it from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

Netbeans setup

- Make sure that Netbeans is installed. You can download it from `http://netbeans.org/downloads/`.
- Open the EPOS project folder in Netbeans.
- Add all libraries from the lib folder to the project.

1.2 Execute the sample simulation

- Start the program
  - Run from command line:
    Navigate to the project directory and execute `java -jar EPOS.jar`.
  - Compile and run in Netbeans:
    Build and run the project with main class `GUIExperiment` (default setting).
- The configuration window opens as shown in Figure 1. Run the simulation by clicking the "Run" button.
- The result window opens and shows details about the simulation as shown in Figure 2. The plot "Global and average local cost" shows the global and local cost in each iteration. The two plots on the bottom show the system state after each

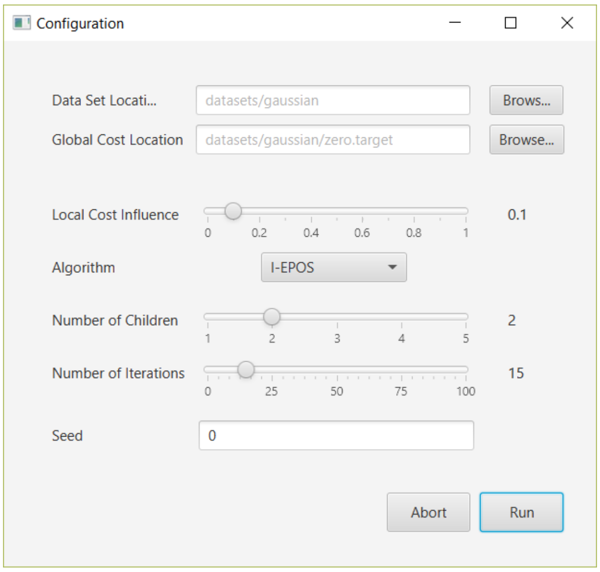

Figure 1: Simulation configuration

iteration. Switch between iterations with the "Next" and "Previous" buttons. The plot "Global Response" shows what the global response, the output of the system, looks like. The plot "Network" shows which agents in the network changed their selections compared to the previous iteration.

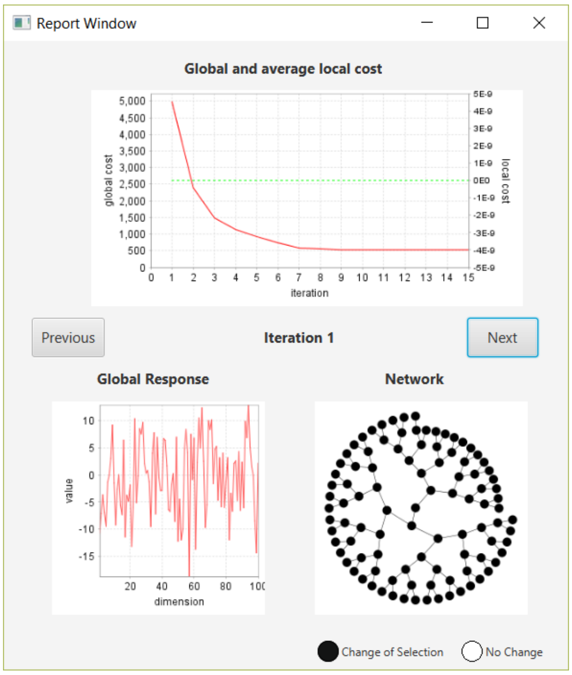

Figure 2: Simulation results

2 Architecture

The software uses the Protopeer framework to simulate agents in a network. The tree is constructed via the TreeGateway package. The agent logic is dependent on the algorithm in use. For I-EPOS, agents are of type `IeposAgent`. For COHDA, agents are of type `CohdaAgent`.

When performing a simulations, all agents are started by Protopeer and TreeGateway constructs the tree. After the tree is built, the agents start to execute the algorithm. Each agent logs different metrics that are specified in the configuration. After the algorithm is finished, all logs are gathered and the aggregated results are presented to the user.

3 Use cases

3.1 Configure the simulation

A simulation example is provided in class `SimpleExperiment`. It contains a main function that specifies all parameters, starts the simulation and presents the results.

Dataset

The `Dataset` interface provides a way to get plans for a given agent via `Dataset.getPlans(int agentIdx)`. There are two classes of datasets implemented:

- `FileVectorDataset` is a dataset that is read from disk. Only the dataset folder has to be specified. The example datasets are located in the directory 'input-data'. Section 3.4 describes the input format for this kind of dataset.

- `GaussianDataset` is a generated dataset where every plan is a vector drawn from a gaussian distribution. The parameters specify the number and dimensionality of the plans as well as mean and standard deviation of the distribution.

The number of agents can be set arbitrarily. However, when using a `FileVectorDataset`, the number of agents has to be below `FileVectorDataset.getNumAgents()`.

For example a dataset of 100 agents taken from the 'bicycle' dataset can be specified as follows:

```
Dataset dataset = new FileVectorDataset("input-data/bicycle");
int numAgents = 100;
```

Cost functions

The global cost function describes what we want to minimize globally. The local cost function describes what each agent wants to minimize locally. `lambda` is the tradeoff between global and local minimization. `lambda = 0` means only global cost is minimized, `lambda = 1` means only local cost is minimized. For global cost functions we can choose any implementation of the interface `CostFunction` in general. However, for gradient descent based algorithms an instance of `DifferentiableCostFunction` is required. A list of possible global cost functions is as follows:

- `DotCostFunction` minimizes the dot product of a given vector with the global response. See Section B.1 how to read the vector from a file. Be aware that the dimensionality of the vector has to match the dimensionality of the dataset. An example use case for this cost function is minimizing monetary cost. The agent plans contain the amount of resources they consume, and the vector passed to `DotCostFunction` describes the price for each resource. As this is a linear cost function, I-EPOS always finds the optimal value in the first iteration.

- `SqrDistCostFunction` minimizes the (squared) distance of the global response to a given vector. See Section B.1 how to read a vector from a file. Be aware that the dimensionality of the vector has to match the dimensionality of the dataset. This cost function tries to make the global response as similar to the provided target vector as possible.
- `VarCostFunction` minimizes the variance of the global response. The effect is that the global response gets stabilized.
- `StdDevCostFunction` minimizes the standard deviation of the global response. For I-EPOS there is no difference between minimizing standard deviation and minimizing variance, as the functions share the same minima.
- `MaxCostFunction` (non-differentiable) minimizes the maximum value of the global response. This function is useful to remove peaks in the global response.

Local cost functions have to implement the `PlanCostFunction` interface. Two functions are implemented:

- `IndexCostFunction` lets agents select plans with a small index. Therefore the plans in a dataset should be ordered in a way that lists preferable plans first.
- `PlanScoreCostFunction` lets agents select plans with a small score. The dataset specifies the score for each plan. See Section 3.4 for details how to specify this information in a dataset.

For example the variance of the global response can be minimized with the following settings:

```
CostFunction globalCostFunc = new VarCostFunction();
LocalCostFunction localCostFunc = null;
double lambda = 0;
```

Network

The network is considered to be a balanced tree with the same number of children for each inner node. The number of children for inner nodes can be specified. Be aware that the runtime of I-EPOS is exponential in the number of children. Luckily the optimization performance of a binary tree is already close to optimal in practice.

For example a binary tree can be specified as follows:

```
int numChildren = 2;
```

Logging

Next we specify what information we want to gather from the simulation. For this task we specify a `LoggingProvider<A>`, where `A` is the class of the agent that is used in the simulation. We then specify all information we want to log by adding `AgentLoggingProviders`.

Each `AgentLoggingProvider` is responsible for reading and presenting one specific type of data. The output is presented after the simulation by calling the method `LoggingProvider.print()`. The following loggers are ready to use:

- `GlobalCostLogger` logs the global cost in each iteration and prints the global cost for each individual iteration averaged over multiple simulations. Note that the sample shown in `SimpleExperiment` only performs one simulation. Multiple simulations can be performed with e.g. different seeds for the agents or different datasets. The only requirement is that the same `LoggingProvider` is used, so all data is gathered in one place.

- `LocalCostLogger` logs the average local cost in each iteration and prints the average local cost for each individual iteration averaged over multiple simulations.

- `TerminationLogger` logs how many iterations it took for the algorithm to terminate. The algorithm is considered terminated if nothing changes between two consecutive iterations.

- `ProgressLogger` prints symbols every few iterations in order to show how far the algorithm has proceeded. It is only useful for large simulations.

- `CostViewer` shows a plot with global and (optionally) local cost values for each iteration in a new window. The logger requires `GlobalCostLogger` to be added to the `LoggingProvider` as well. If the `LocalCostLogger` is present, the local cost is also shown in the plot.

- `GraphLogger` shows a graph of the network at a given iteration in a new window. With the arrow keys you can switch between different iterations. Each agent is represented in a certain color. The color code depends on the specified type `GraphLogger.Type`. The following types are available:

  - `Change` marks each agent that changed its selection in the previous iteration as black and all agents without change as white.
  - `Index` colors each agent based on the index of the selected plan. Agents that selected the plan with minimal index are colored white and agents that selected the plan with maximal index are colored black.

- `FileWriter` writes the log to the specified directory once `LoggingProvider.print()` is executed.

- `FileReader` reads the log from the specified directory once `LoggingProvider.print()` is executed. This logger can be used to show results from a previous simulation that were stored with `FileWriter`. A sample application of `FileReader` can be seen in `ReplayExperiment`.

For example the global response can be logged and presented in a new windwow as follows:

```
LoggingProvider loggingProvider = new LoggingProvider();
loggingProvider.add(new GlobalCostLogger());
loggingProvider.add(new CostViewer());
```

Algorithm

Finally, the optimization algorithm has to be specified. The algorithm is determined by the type of `Agent` that is used.

- `IeposAgent` has quite a few options that were part of the research. We need to specify the number of iterations the algorithm should perform. For problems with less than 1000 agents the (local) optimum is usually found with less than 20 iterations. In addition we can specify a `PlanSelector`. The default is `IeposPlanSelector`. As part of the research that was done for I-EPOS, gradient descent motivated plan selectors were also developed. However, in general they have lower performance than the default.

- `CohdaAgent` is an algorithm that was used as a baseline for I-EPOS. We only need to specify the number of iterations for this algorithm. Limitations: First, `CohdaAgent` does not support local cost. Therefore `LocalCostLogger` cannot be used. Second, the algorithm starts with an incomplete global response that is missing data from some agents. It takes `log(numAgents)/log(numChildren)` iterations for the global response to be complete. Third, even though COHDA does not require the network to be a tree, only tree networks can be simulated with this software.

In order to use the I-EPOS algorithm, the system should consist of `IeposAgent` nodes. This can be specified via the following function:

```
Function<Integer, Agent> createAgent = (Integer agentIdx) -> {
   return new IeposAgent(
      numIterations,
      dataset.getPlans(agentIdx),
      globalCostFunc,
      localCostFunc,
      loggingProvider.getAgentLoggingProvider(agentIdx, 0));
};
```

Run the simulation

Now that the configuration is complete, the simulation can be started as follows:

```
IeposExperiment.runSimulation(
   numChildren,
   numIterations,
   numAgents,
```

```
   createAgent);
loggingProvider.print();
```

3.2 How to store evaluation results

Evaluation results can be stored using the class `FileWriter`. Once the print command is executed on the `LoggingProvider` instance, the log is written to the specified file. With `FileReader`, the written file can be read again. See `ReplayExperiment` as an example how to read a log file.

For example, logging the global cost and storing it in the file 'mylog.log' can be specified in the configuration as follows:

```
...
LoggingProvider loggingProvider = new LoggingProvider();
loggingProvider.add(new GlobalCostLogger());
loggingProvider.add(new FileWriter("myLog.log"));
...
```

Reading the logged data and displaying it again can be done in a separate program:

```
public static void main(String[] args) {
   LoggingProvider loggingProvider = new LoggingProvider();
   loggingProvider.add(new FileReader("myLog.log"));
   loggingProvider.add(new CostViewer());

   loggingProvider.print();
}
```

3.3 Write a new cost function

Write a new class that extends the abstract class `CostFunction<DT>` or `DifferentiableCostFunction<DT>` where `DT` is the datatype that this cost function should operate on. The function `CostFunction.calcCost(DT value)` should compute the cost of the given value. For differentiable functions we also need to implement `DifferentiableCostFunction.calcGradient(DT value)` that should return the gradient of the function at point `value`.

For example, a cost function that computes the cost as the smallest value of a vector could be implemented as follows:

```
public class MinCostFunction extends CostFunction<Vector> {
   public double calcCost(Vector vector) {
      return vector.min();
   }
}
```

3.4 Add a new dataset

One way of adding a new dataset is to use the existing class `FileVectorDataset` to read a custom dataset from the dataset directory. The dataset is a directory containing one file for each agent. The files should be named `agent_<id>.plans`, where `<id>` is the id of the agent, starting from 0 upwards. Each file should be a text file containing one row for each possible plan the agent can choose. A plan has the following layout: `<score>:<vector>`. The score is a double value that describes the cost this plan imposes for an agent. It can be used for local cost minimization[1]. The vector is a comma separated list of double values.

It is also possible to code a new dataset. The only requirement for the dataset is to generate a list of plans given the index of an agent. `Dataset` is a handy interface that can be used to implement a new dataset. While the sample datasets all use vectors as datatype, the new dataset can use a custom datatype.

The following example dataset generates the plans {`1,0,0`},{`0,1,0`} `and` {`0,0,1`} for each agent:

```
public class MyDataset implements Dataset<Vector> {
   private List plans = new ArrayList();
   public MyDataset() {
      for(int i = 0; i < 3; i++) {
         Vector v = new Vector(3);
         v.setValue(i, 1);
         plans.add(new Plan(v));
      }
   }
   public List<Plan<Vector>> getPlans(int agentId) {
      return plans;
   }
}
```

3.5 Add a new datatype

Write a new class that implements the interface `Datatype`. The functions of the interface `DataType` were designed work for vectors. So when implementing those functions, be aware that the semantic should be as it would be for vectors.

To use the new datatype, we also need to implement a new dataset and a new cost function that can handle the new datatype.

[1] Set `lambda=0` if the score should be ignored.

A Glossary

B Utility functions

B.1 Reading a vector

A vector can be read from a file via `VectorIO.readVector(File vectorFile)`. The file is assumed to be text file, containing a comma-separated list of double values that make up the vector.