iNVIDIOSO 1.0

Federico Campeotto

1 System Requirements

iNVIDIOSO takes advantage of the GPU architecture to perform parallel computation. However, it is possible to install a sequential version of the solver by choosing the appropriate option during installation.

1.1 C++11

iNVIDIOSO is written in C++11 and CUDA. Therefore, in order to compile the source code of the solver the system must be equipped with a **C++11** compiler.[1] To check whether the compiler supports C++11, do what follows:

- Linux
  Type the following in the terminal:

  ```
  $ g++ –version
  ```

  C++ is (partially) supported from version 4.3. The solver has been tested with version 4.7. You may want to update your current compiler version as follows:

  ```
  $ sudo apt-get update
  $ sudo apt-get install gcc-4.7
  $ sudo apt-get install g++-4.7
  ```

- MacOS X

  ```
  $ clang –version
  ```

  The terminal should print something like the following:

  ```
  Apple LLVM version 6.0 (clang-600.0.56) (based on LLVM 3.5svn)
  Target: x86_64-apple-darwin14.1.0
  Thread model: posix
  ```

[1] NVIDIOSO was tested on gcc4.7, so I recommend to use gcc4.7 or higher. If an older version is present on the system, please, change the *c++11* variable into *c++0x* in the *install.csh* script.

Otherwise, you may want to install the XCode package. XCode is available from the Mac OS install DVD, or from the Apple Mac Dev Center. Another option for installing clang and the C++11 compiler on MacOS X is to install the Xcode Command Line Tools:

```
$ xcode-select –install
```

Finally, g++ can be also installed using *home-brew* (see the notes printed during the installation of iNVIDIOSO).

1.2 CUDA

In order to use the computation power of the GPU, the solver must be compiled using ***nvcc* CUDA-6.5**. The installation does this automatically after setting the path where the nvcc compiler resides. First, check if you have nvcc already installed and its path set into the current environment:

```
$ nvcc –version
```

If you see something like the following:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2014 NVIDIA Corporation
Built on Thu_Jul_17_19:13:24_CDT_2014
Cuda compilation tools, release 6.5, V6.5.12
```

then nvcc is installed and you don't need to change the path in the installation script.[2]

If you receive an error message, then you need to download and install nvcc from `http://developer.nvidia.com/cuda-downloads` (skip this passage if you are sure that nvcc is already present in your system). Moreover, you need to set the global path of the nvcc binary in the installation script "install.csh":

```
set NVCC =/machine_name/cuda-6.5/bin/nvcc
```

Note that your system could have a nvcc release version lower than 6.5. In such a case, you should update nvcc following the above instructions and export the path in the PATH variable.

2 Obtaining iNVIDIOSO1.0

iNVIDOSO can be downloaded from a *github* repository by clicking on the *download ZIP* button. If you want to contribute to the project, you should have git

[2] You may need to set the path even if nvcc is recognized as command in your current shell. Check error messages during installation.

installed in your system the system. To check if you have git on your system, type the following in your terminal:

```
$ git –version
```

If you receive an error message, then you can choose between two options: (1) create a github account on `github.com` and then download a zip version of the solver from `github.com/FedericoCampe8/NVIDIOSO`, or (2) install git on your system and use a cloned version of the solver to collaborate in its development. To install git in your system, type the following command in your terminal:

```
$ sudo apt-get update
$ sudo apt-get install git
```

Please, note that developers should collaborate to the project using git and pushing their extensions on github server. Once you have git properly installed in your system you can clone the project, modify it, and then commit the new changes on your local version and/or push them on the server. For new developers I highly recommend to create a github account and **fork** the project by clicking the *fork* button on and then clone the repository.

In the next section I report a simple working example of how to use git to modify and update files on the master branch (i.e., on the server). However, I strongly encourage to read the git tutorial present on git webpage (`http://git-scm.com/documentation`).

2.1 Simple working example

Suppose that two developers, John and Jessica, want to work together with a shared repository that is on some server. John starts by cloning the repository on his local machine, modifying some files, and committing the changes:

```
# John's machine
$ git clone john@githost:simplegit.git
$ cd simplegit/
# Some modifications to the code here
$ git commit –am 'fixed bug abc'
```

Jessica does the same with her local copy:

```
# Jessica's machine
$ git clone jessica@githost:simplegit.git
$ cd simplegit/
# Some modifications to the code here
$ git commit –am 'add some descriptions'
```

Now Jessica pushes her work up to the server

```
# Jessica's machine
$ git push origin master
```

In the mean time, John tries to do the same

```
# John's machine
$ git push origin master
```

But he receives an error message since the server has a most updated copy of the project (updates coming from Jessica). Therefore, John has to first fetch the new updates, and then merge them on his local copy:

```
# John's machine
$ git fetch origin
$ git merge origin/master
```

Then he can update his version on the server:

```
# John's machine
$ git push origin master
```

In the mean time, Jessica has done some other local work she wants to update on the server:

```
# Jessica's machine
# First she creates a new branch from the master branch
$ git branch fixBug
# Then she works on code.
# When done, she merges the work with her current master branch
$ git fetch origin
$ git checkout master
$ git merge  fixBug
# Now she updates the code on the server
$ git merge origin/master
$ git push origin/master
```

The above example presents a normal working flow using git. In particular, I recommend to create a new branch for every new update/modification on the local master branch:

```
$ git branch hotFix
```

After working on that branch, it should be merged with the master and deleted:

```
$ git checkout master
$ git merge hotFix
$ git branch -d hotFix
```

To see the status of your git repository (including branches) use `git status`. To clone the repository via SSH (recommended) use the following address: `git@github.com:FedericoCampe8/NVIDIOSO.git`. To generate a SSH public key follow the instructions present in `http://git-scm.com/book/it/v2/Git-on-the-Server-Generating-Your-SSH-Public-Key`. Then, email your public key to `fede.campe@gmail.com` to be added as a developer.

Some notes:

- Fork the project on your personal github account would be a better idea instead of cloning it directly from the server
- Branch from master every time you want to modify the code with important changes
- Remember to fetch the updates from the server before pushing them on github
- To remove a file/folder use `git rm <file>`

3 Installation

To install the solver, go into your NVIDIOSO folder (e.g., *iNVIDIOSO-master/NVIDIOSO*) and run the following command:

```
$ ./install.csh
```

This will build iNVIDIOSO. During the installation you will be asked to choose whether to install the sequential version or the parallel version of the solver. Type “c” for installing the CPU sequential version and “g” for the GPU parallel version. Depending on the system you are using, during the installation you may receive different advices. The installation process will generate a file named *make.inc* and a file with extension *.log*, in case of failure.

4 Using the Solver

The solver can be run using the following command

```
$ ./nvidioso -i infile [options]
```

where *infile* is the input file. For example:

```
$ ./nvidioso -i test/nqueens.fzn -v
```

runs the solver with the nqueens.fzn file as input and the verbose $(-v)$ option. To see the list of options available to the solver, use the -h option:

```
$ ./nvidioso -h
```

5 Structure of iNVIDIOSO

iNVIDIOSO is written in C++11 and it is based on two main components:

1. A *Data Store*: it holds all the information about variables and constraints;
2. A *(CP) Solver*: it uses the data retrieved from the data store to run its internal search strategies to find solutions.

5.1 Data Store

A Data Store is constructed from a model written as a text file and passed to its constructor. The internal parser reads the file and initializes all the data structures and objects needed during the search phase and demanded by the (CP) Solver. Let us observe that iNVIDIOSO does not parse only *FlatZinc* files but it can read any constraint model (e.g., *GeCode* models) by deriving the parser class according to the different modeling language. For example, the `cp_store` class deriving from `DataStore` (see `inlcude/base`) contains a pointer to an instance of the parser to use for reading the model from the file. This class contains a member function called "init_model" which reads the file and uses the parser to tokenize the strings and generate the corresponding objects in terms of *variables*, *constraint*, etc.

Another important element of the Data Store is the *CPModel* (see `include/core`). The CPModel is a class representing a general CP Model which includes *variables*, *constraints*, a *search engine* and a *constraint store*. These information is provided by the cp_store during initialization. The CPModel also attaches the constraint store to all the variables in the model following an Observer/Subject pattern: every time the domain of a variable is modified, a notification is sent automatically to the constraint store.

5.2 (CP) Solver

The CP Solver (see `include/base`) is a class that implements a CP solver deriving from *Solver*.

The Solver class is an interface for defining solvers. It provides methods such as *add_model* or *run* for adding models and running them respectively. Therefore, the whole solver is not bounded to a specific type of solver but it can run different solvers (e.g., a Planner), provided the right instance of a class derived from Solver which implements its interface.

CP Solver The CP Solver class is a class derived from Solver for solving CP instances. In particular, this class runs a CP model by first creating a constraint graph from the model and then invoking the *labeling* method of the search engine associated with the CP model (see `src/base/cp_solver.cpp`). Note that this class can run different CP models at "the same time", meaning that is possible to upload different CP instances and solve them sequentially[3].

[3] As a future work we are planning to implement a version of iNVIDIOSo where different instances are run in parallel on different machines or different GPUs

6 Core components

iNVIDIOSO uses several core component (i.e., classes). These classes are declared in `include/core`. The most important classes are:

- *constraint_store*: interface for a constraint store. Probably, the most important method of this class is the *consistency* method which propagates constraints and returns *True* if all propagated constraints are consistent or *False* otherwise.
- *variable*: abstract class representing the variables of the solver. These variables have a list of constraints (constraints for which they are involved in) and a pointer to the constraint store to notify whenever their domain is changed. Therefore, variables have a *DomainIterator* member which represents an *iterator* to the associated domain and which is used to perform actions on it.
- *domain*: abstract class representing a domain. This class does not specify how a domain should be implemented. Instead, it specifies a set of methods to define in each class deriving from it. For example, see *int_domain*.
- *memento*: this is a class implementing a memento pattern for "Backtrackable" objects. A class which implements a BacktrackableObject (see *int_variable*) can store its state on a stack and retrieve the same state later during the computation. This is how a backtrack search strategy is implemented.

7 Constraints

Constraints are declared in `include/constraint`. A constraint is an object which is bounded to a set of variables and which perform some actions on these variables, changing their internal state. Two important methods are *consistency* and *satisfied*, which propagates the constraint and check for its satisfiability respectively. In iNVIDIOSO1.0 the constraints are those defined in the FlatZinc specification. In particular, the class *fzn_constraint* derives from Constraint an defines all the FlatZinc constraints. In turn, each FlatZinc constraint derives from *fzn_constraint* and implements the specific methods declared in the interface. For example, see `include/constraints/int_ne.h` and its definition in `src/constraints/int_ne.cpp`.

8 Search

Search engines are declared in `include/search`. A *search engine* is an interface which represent a general search engine. It defines several methods which are used by the solver to perform search. For example, *SearchEngine* defines the *labeling* method which perform the actual search. First it sets up the internal attributes of the search. Then, it calls the labeling function with argument specifying the index of a non grounded variable.

The class *DepthFirstSearch* is an example of a search strategy implemented as a derived class from SearchEngine. From the implementation point of view, different heuristics can be defined by deriving from the *Heuristic* class. This class is an interface used by the search engine classes in order to select the next variable to label (*get_choice_variable* method) and the value to assign to it (*get_choice_value* method).

9 Other components

In what follows I briefly describe the other main components of the system:

- *FZ_parse*: parse for FlatZinc models. It describes how to tokenize FlatZinc files and which tokens should be produced by this tokenization.
- *exception*: it declares the exceptions that can be raised during the computation. This is an important class and developers must use exceptions and catch them to properly write the code and later debug it.
- *base*: general classes. There are factory method, identifier generators, a class for statistic analysis, a logger and other useful classes used by iNVIDIOSO1.0.

10 How to

The above sections briefly describe the core components of the system. The developer should refer to the html API page where it is possible to file the complete list of classes and their hierarchy. Moreover, there is a description for every class and every method of that class. These descriptions are present in the folder "iNVIDOSO/html". Open a browser and type

```
<path_to_invidioso>/iNVIDIOSO/html/index.html
```

to navigate through the API using your browser.

10.1 Where do I start?

I personally like a top-down approach when I'm trying to understand the code someone else has written. So, I will briefly describe the code starting from the main function, i.e., from the file *main.cpp*. If you open *main.cpp* you will see the main function which does six "main" things (don't consider logger and statistics messages for now):

1. It creates a singleton object *InputData*: this is nothing more than a parser for the input given by the user;
2. It creates a singleton object *d_store* of type *DataStore*: this object represents the "store" containing all the information about the model(s) to solve (i.e., variables, constraints, and search strategies);
3. It loads the model(s) into the store:

   ```
   d_store->load_model()
   ```

4. It initializes the model(s) by filling the data structures within the DataStore object:

   ```
   d_store->init_model();
   ```

5. It generates a new solver of type *CPSolver* for solving the model previously initialized:

   ```
   CPSolver * cp_solver = new CPSolver( d_store->get_model() );
   ```

6. It solves the model(s):

   ```
   cp_solver->run();
   ```

In what follows I will give a brief description of almost all of the above steps. I will skip the *InputData* class since it is a really straightforward parser class.

Model loading and initialization. Model loading and initialization are done by methods declared in the CPStore class. CPStore is a subclass of DataStore which specializes the DataStore for *Constraint Programming* models. The definition of the methods *load_model()* and *init_model()* can be found in *cp_store.cpp*. The *load_model* function creates a parser for *FlatZinc* style input files (a different parser can be provided) and iterates through the input file untile the parser can find tokens.[4] These tokens are stored in the parser and are ready for be given to the store and converted into more “high-level” object used during the solving process.

The *init_model* method creates a new CPModel object and fills it with variables, constraints, the constraint store, and a search engine, according to the parsed model. This is done by creating a *generator* object which task is to convert tokens given by the parser to solver’s object of the proper type (just look at it as a general abstract factory class).[5] Note that a CPModel object represents the a model to be solved by the solver. A CPStore, instead, represents a “global” container of information for one or more (CP) models to solve.

Solving a model. This subsection is a draft. Briefly, the “important” methods are defined in *cp_solver.cpp*. In particular, the *run()* method runs all the models added to the CPSolver instance executing the following steps:

1. It creates the *constraint graph* (i.e., it attaches constraints to variables);
2. It initializes the constraint store for an initial propagation (before starting the search process);
3. It attaches the constraint store to each variable so any variable can notify the constraint store upon any change on its domain;

[4] In this case, a token is an object encapsulating the information of the various components of a given model, e.g., variables, constraint, etc.

[5] The methods with the *add_* prefix are used to fill vectors representing the state of a CPModel instance.

4. It invokes the labeling function of the search engine which *labels* the variables and performs backtrack according to a given search strategy.

The *run()* method calls the run_model method on each model (i.e., models saved in the vector of CPModels“_models”). Let’s consider a specific model, e.g., the CPModel *_models[0]* (note that the vector _model contains pointer to the CPModel objects). The run_model method perform the following main steps:

1. It creates the constraint graph for the model by attaching each constraint to the variables in its scope (create_constraint_graph()):

```
// See cp_model.cpp
for ( auto c : _constraints )
        c->attach_me_to_vars();
```

   This allows a variable to notify the constraints it is involved in whenever its domain changes (see Observer pattern).
2. It initializes the constraint store (init_constraint_store). This fills the constraint store with all the constraints in order to perform an initial propagation before the search phase starts. The initial propagation of constraints is needed in order to prune domains or to find unsatisfiable models as soon as possible (e.g., $x = 2; y = 2; x \neq y$);
3. It attaches the constraint store to each variable in the model (attach_constraint_store()):

```
// See cp_model.cpp
for ( auto var : _variable)
        var->attach_store( _store );
```

   This is done for two main reasons: first, it decouples the variables to the constraint store, meaning that we can change the constraint store during the search without compromising the status of the variables. Second, a variable can notify the constraint store whenever its domain changes (see Observer pattern). In particular, this allows the variables to add the constraints they are involved in into the queue of constraints of the constraint store for the following propagation phase.
4. It starts the search by calling the method ”labeling” defined for each (valid) search engine:

```
(model->get_search_engine())->labeling();
```

The first 3 steps are pretty straightforward, so we will focus on the last point: the labeling phase.

Labeling. The labeling method allows the solver to perform the actual search. The search phase is carried on by a SearchEngine object (see *search_engine.h*) which must implements all the pure methods of the abstract class SearchEngine. A search engine is composed by several objects:

– A *constraint store* to invoke for propagating constraints during search;

– A *heuristic* that defines the order of variables to label and the values to assign to them;
– A *solution manager* that keeps track of all solutions found during the search and other statistics;
– A *backtrack manager* that allows the search to *backtrack* to previous states for backtrack strategies (e.g., depth first search). Backtrack manager is implemented using the *memento* pattern.

To check how a search engine really works, let's consider the depth first search engine (see depth_first_search.cpp) and, in particular, the *label* method, invoked by *labeling()*. The label method takes the current level of the search tree as input parameter (var_idx) and calls itself recursively until either a(ll) solution(s) is(are) found or the model is proven to be unsatisfiable. Let's us observe that the index given to the label method is not the index of a variable among the array of variables to label. Instead, it represents the level of the search tree and only as a "side effect" it corresponds to a specific variable.

The first thing that the label method does is to propagate the constraints by invoking the *consistency* method of the constraint store:

```
consistent = _store->consistency();
```

If the store is not consistent (e.g., a variable has an empty domain), then the search returns *false* and, in case, it backtracks. Otherwise, the propagation was successful and the search can continue by labeling the next variable. In particular, the new (pointer to a) variable to label is chosen by the heuristic

```
var = _heuristic->get_choice_variable ( var_idx );
```

as well as a value for it

```
value = _heuristic->get_choice_value ();
```

and the labeling is performed by shrinking the domain of the selected variable to the chosen value:

```
(static_cast<IntVariable*>(var))->shrink ( value, value );
```

The search then continues with a recursive call:

```
int next_index = _heuristic->get_index();
...
consistent = label ( next_index );
```

If the result of a recursive call is *true*, all the children of the current node of the search tree are explored (and consistent) and the search returns. If, instead, the result of a recursive call is *false*, the current assignment of value to var leads to a failure somewhere in the subtree. This value must be removed from the domain of the current labeled variable and another labeling must be performed:

```
(static_cast<IntVariable*>(var))->subtract ( value );
...
consistent = label ( var_idx );
```

Note that before doing the next labeling the status of the search phase must be restored, i.e., a backtrack must be performed:

```
_backtrack_manager->remove_level ( _depth );
```

Whenever a new labeling is not possible the search returns *false*.

For a detailed explanation, please contact `campe8@nmsu.edu`.

10.2 Notes for developers

These are some notes for developers:

- Write the code as clear as possible
- Use “_variable” for private variables where “variable” is the name of the private variable
- Comment every method when you define a new class as well as a brief description of the class at the top of the file.
- The comments should be parsed by Oxygen, therefore use “/** – comments here – */” for comments spanning on multiple lines read by Doxygen. Use “//!” single line comments.
- Use “@param parameter” for describing the list of parameters of a function, “@return value” to describe the return values, and “@note notes” if you want to add an extra note.
- Use design patterns.
- Prefer decoupling of classes.
- Programming to an Interface, not an Implementation
- Favor object composition over class inheritance
- Use C++11 and no old style C programming. For example:
  - Use *new* and *delete* instead of *malloc* and *free*.
  - Use C++ *string* instead of *char * string*.
  - Prefer *unique_ptr* to *shared_ptr* and avoid to use standard pointers.
  - Prefer *vector(s)* instead of standard array(s).
  - Use the C++(11) *Standard Template Library* (*STL*) instead of reinventing the wheel. STL has built in data structure and algorithms that can satisfy the majority of the needs (e.g., use *(unordered_)map* for hashing).
  - Comment your code.
  - Refer to the C++ guide, e.g., see `http://en.cppreference.com/w/` or `http://www.cplusplus.com/`.

Thanks for reading this page. If you' have any further questions, please don't hesitate to contact campe8@nmsu.edu.