Programming Exercise: a Rounding Heuristic

Rounding heuristics try to set integer variables which take a fractional value in a given LP solution x^* to a feasible integral value. A simple criterion for the direction that is suitable for rounding are the locking numbers of a variable. For instance, given a linear system $Ax \leq b$, the number of down-locks of a variable is the number of linear constraints in which the variable appears with a negative coefficient. The number of up-locks is the number of inequalities in which the variable appears with a positive coefficient. If a variable x_j has no down-locks (up-locks), then rounding down (up) the value of x_j maintains linear feasibility.

The task of this programming exercise is to implement a *Rounding Heuristic*. This heuristic will round variables if they have a direction without rounding locks. If this holds for all fractional variables, the heuristic will produce a feasible solution, otherwise it should abort.

Getting started

In order to be efficient, we will only use one working solution for all calls of the heuristics. Furthermore, we only want to compute the number of "easy-roundable" variables once. Therefore, we use a global data structure, the `SCIP_HEURDATA`. It has to be allocated in `SCIP_DECL_HEURINIT` and freed in `SCIP_DECL_HEUREXIT`.

1. Go to the directory `src/scip` and copy the two template files `heur_xyz.c` and `heur_xyz.h` to files named `heur_lockrounding.c` and `heur_lockrounding.h`, respectively. Open the new files with a text editor and replace all occurrences of `xyz`, `Xyz` and `XYZ` by `lockrounding`, `Lockrounding` and `LOCKROUNDING` respectively.

2. Fill in a `SCIP_SOL* workingsolution` and an `int nroundablevars` into the global structure `SCIP_HeurData`.

3. In `SCIPincludeHeurLockrounding()`, allocate memory for your heuristic data using `SCIP_RETCODE SCIPallocMemory(SCIP* scip, HEURDATA** heurdata)`.

4. Implement `SCIP_DECL_HEURINIT`. Get your heuristic data with `SCIP_HEURDATA* SCIPheurGetData(SCIP_HEUR* heur)`, and set up the working solution in your heuristic data with `SCIPcreateSol(SCIP* scip, SCIP_SOL** sol, SCIP_HEUR* heur)`. Finally, count the number of roundable variables and store the information in your data structure.

5. Implement `SCIP_DECL_HEUREXIT`. Get your heuristic data, free the working solution with `SCIP_RETCODE SCIPfreeSol(SCIP* scip, SCIP_SOL** sol)` and the heuristic's data structure with `SCIPfreeMemory(SCIP* scip, SCIP_HEURDATA** heurdata)`.

Hints:

- `SCIP_DECL_HEURINIT` is a macro which is defined in `src/scip/type_heur.h`. It states that `SCIP_DECL_HEURINIT(heurInitLockrounding)` is equivalent to `SCIP_RETCODE heurInitLockrounding (SCIP* scip, SCIP_HEUR* heur)`.
- `SCIP_RETCODE SCIPgetVarsData(SCIP* scip, SCIP_VAR*** vars, int* nvars, int* nbinvars, int* nintvars, int* nimplvars, int* ncontvars)` provides you information about the problem's variables.
- `SCIP_Bool SCIPvarMayRoundUp(SCIP VAR* var)` counts the number of up-locks for a given variable and returns whether it can be rounded up. There is also a function `SCIP_Bool SCIPvarMayRoundDown(SCIP VAR* var)`

Implementing the fundamental callbacks

The fundamental callback of a heuristic is its execution method.

1. Your heuristic may be called for every LP solution, so the best timing probably is `SCIP_HEURTIMING_DURINGLPLOOP`.
2. Get the fractional variables of the current LP solution and start your heuristic if there is an appropriate number of them. This is achieved by using the function `SCIP_RETCODE SCIPgetLPBranchCands(SCIP* scip, SCIP_VAR*** lpcands, SCIP_Real** lpcandssol, SCIP_Real** lpcandsfrac, int* nlpcands, int* npriolpcands, int* nfracimplvars)`.
3. `SCIP_RETCODE SCIPlinkLPSol(SCIP* scip, SCIP_SOL* sol)` "copies" the current LP solution to your working solution.
4. Now iterate over all fractional variables, if possible round them by changing their solution value with `SCIP_RETCODE SCIPsetSolVal(SCIP* scip, SCIP_SOL* sol, SCIP_VAR* var, SCIP_Real val)`. If you find a variable which is neither down- nor up-roundable, abort.
5. If you were able to round all variables, hand over the solution by `SCIP_RETCODE SCIPtrySol(SCIP* scip, SCIP_SOL* sol, SCIP_Bool checkbounds, SCIP_Bool checkintegrality, SCIP_Bool checklprows, SCIP_Bool* stored)`. If SCIP accepted your solution, set the result pointer to `SCIP_FOUNDSOL`. If you searched for a solution, but did not succeed, it should be `SCIP_DIDNOTFIND`, otherwise `SCIP_DIDNOTRUN`.

   Hint: If you do not want to carry around the heuristic's data structure all the time, just set a local pointer to your working solution.

```
SCIP_HEURDATA* heurdata;
SCIP_SOL* workingsolution;
heurdata = SCIPheurGetData(heur);
workingsolution = heurdata->workingsolution;
```

Including the heuristic

There are three steps needed to include the heuristic to scip as a default plug-in.

1. Add `scip/heur_lockrounding.o` to `SCIPPLUGINLIBOBJ` in the `Makefile`.
2. Include `scip/heur_lockrounding.h` in `src/scip/scipdefplugins.h`.
3. Finally, call the `SCIP_RETCODE SCIPincludeHeurLockrounding(SCIP* scip)` method in `src/scip/scipdefplugins.c`.

Additional exercise

It may not be necessary to break just because there are some positive locking numbers. Think about a possible heuristic that might handle these situations.

Additional information

- Analogously to your plugin specific data structures you may set up and destroy arrays. The functions `SCIP_RETCODE SCIPallocBufferArray(SCIP* scip, <yourarray>*, int* length)` and `SCIP_RETCODE SCIPfreeBufferArray(SCIP* scip, <yourarray>*)` will provide the required functionality.
- You may add your own user parameters to your plugin. This is useful for testing. Check out methods called `SCIPadd<Datatype>Param`. They are called in the `SCIPinclude<YourPlugin>()`. E.g., `SCIPincludeHeurLocalbranching()` adds a modifiable parameter via calling `SCIPaddIntParam(scip, ‘‘heuristics/localbranching/neighborhoodsize’’, ‘‘radius (in Manhattan metric) of the incumbent’s neighborhood’’, &heurdata->neighborhoodsize, FALSE, 18, 1, INT_MAX, NULL, NULL)`.
- By including the parameters it is then possible to change these in the interactive shell: “set heur loc nei 42”, and store the settings: “set diff settings/mysettings.set”.

Testing the plugin

1. To begin testing, open the interactive shell and turn off all heuristics. This is done by the command "set heur emph off". Store the settings by the command "set diff settings/heuroff.set".

2. Then use the "set" command in the interactive shell to turn on the heuristic that has just been implemented. This is achieved similar to local branching neighbourhood size presented above. You must have implemented a parameter that permits the heuristic to be switched on or off.

3. Once the implemented heuristics has been switched on, save the parameters using "set diff settings/roundon.set".

4. Now, we wish to test SCIP without heuristics and with the implemented heuristics. First call "make TEST=short TIME=60 SETTINGS=heuroff test" (testing without heuristics) and the results will be saved in "check/results/check.instances.<...>.heuroff.res".

5. Then call "make TEST=short TIME=60 SETTINGS=roundon test" (testing the lock rounding heuristic). Similarly the results will be saved in "check/results/check.instances.<...>.roundon.res".