Report implementation exercise 1

Heuristic Optimization

Arno Moonens
VUB, Student nr.: 500513
arno.moonens@vub.ac.be

1. Implementation

I used the programming language `C` for this implementation exercise and used the example file to built upon.
The program starts by constructing a `Solution`. This is a `struct` containing all variables that are different for each solution for the set covering problem, e.g. the cost of the solution, an array that says which sets are used, …
To build the solution, the `execute` function is called. First, we need to choose a set. The way of choosing a set depends on the constructive method that was chosen by passing an argument to the program (see Section 3).
For *CH1*, we keep choosing an element randomly until we find one that isn't covered yet. After that, we randomly choose a set that covers that element. For the other constructive methods, we loop over the sets to find the *best* one. Which one is the best varies on the method chosen and is implemented the same way as it was described in the exercise description. The set can only be chosen if it covers elements that weren't covered already.
After a set was chosen, we add it to the current solution, thus also increasing the solution's cost by the cost of the chosen set. This process of choosing a set and adding it to the solution is repeated until all elements are covered.

When it is signaled using parameters that redundancy elimination, best improvement or first improvement needs to be applied, we first sort the sets by their cost in decreasing order. This is done in order to quickly find the set in the solution with the highest cost without having to see every set, thus improving the program's execution speed. By sorting, we get an array with the indexes of the sets in decreasing cost.
In case we need to apply redundancy elimination, we try to remove each set. We do this in decreasing order of cost using the mentioned array of sorted costs. For each set, we first go over each element and see if it covers the element. If, in the current solution, the element is covered by the set but it isn't covered by any other set, we can't remove the set because otherwise the element wouldn't be covered anymore. If, on the other hand, all elements were checked and they are either not covered at all by the set or covered by sets other than the set

we're currently looking at, we can safely remove the set from the solution. This removal includes, amongst other things, the decrease of the solution's cost by the cost of the set that has to be removed.
For best and first improvement, we keep looping until no improvement has been found. In both algorithms, we copy the currently best solution, remove the set in the solution with the `i`'th highest cost. Because we don't do any checks (other than the requirement that the set currently needs to be used in the solution), it is possible that in the solution after the removal, not all elements are covered anymore. This is solved by applying the already explained `execute` function with the Adaptive cover cost-based greedy heuristic as constructive method. The cost may also be different from the solution that we originally copied. In the `best_improvement` function, we try each removal-and-rebuild (remove the highest-cost set and rebuild the solution, remove the second highest-cost set and rebuild the solution,…) and retain the solution with the lowest total cost. If this cost is lower than the one of the solution we already had (from the solution we copied each time), we apply redundancy elimination to further reduce the cost, assign the found solution as the new best one and start a new loop of removing and rebuilding. If the solution wasn't better, the algorithm stops, with the passed double-pointer now eventually pointing to the (possibly) newly found solution.
The first improvement algorithm works a bit different. We also start each time by copying the solution that is currently the best, removing a set from the solution and rebuilding the solution. If however, the new solution is better than the one that is currently the best, we immediately apply redundancy elimination and assign the new solution as the best one. In the next copy-remove-rebuild, this new solution is then immediately used in order to try to find a better one. When all sets were tried and no new solution was an improvement on the best one, the algorithm stops. Otherwise, we start again on a copy of the currently best one, by first removing the set with the highest cost and by then rebuilding the solution.

2. File structure

Different functionalities are located in separate files, which are the following:

- `main.c`: Parses arguments passed to the binary and calls the rights constructive heuristic, redundancy elimination, etc.
- `instance.h` and `instance.c`: Has code to build a representation of a problem instance using a file and to print information about the instance.
- `solution.h` and `solution.c`: Info about a solution and functions to handle a solution, e.g. adding a set, applying redundancy elimination, getting the number of uncovered elements,…
- `complete.h` and `complete.c`: Contains code to build a solution using *CH1*, *CH2*, *CH3* and *CH4*.
- `improvement.h` and `improvement.c`: Includes functions to apply first im-

provement and best improvement.
- `utils.h` and `utils.c`: Contains help functions unrelated to specific functionalities.
- `run.sh`: Runs all the experiments and prints results to files.
- `analysis.r`: Reads the result files and analyzes the results.

3. Usage

First the `C` code needs to be compiled with a `C` compiler. On *Mac OS X* (with *Apple Developer Tools* installed) and *Ubuntu Linux* (with *gcc* installed), this can be done using the following command (when in the directory with the `C` source code):

```
gcc main.c utils.h utils.c complete.h complete.c solution.h solution.c
instance.h instance.c improvement.h improvement.c
```

This should result in a binary file called `a.out`.
The resulting binary can then be called by some parameters. These are the following:

- `--seed SEED`: The `SEED` to use for the random number generator. If no value is provided, `1234567` is used as a seed.
- `--instance path/to/instance`: The path to the instance file that needs to be used. This value is required.
- `--ch1`, `--ch2`, `--ch3` or `--ch4`: Constructive method to be used to build the initial solution. Exactly one of these has to be provided.
- `--re`: This optional parameter signals that redundancy elimination needs to be applied after the initial solution is constructed.
- `--bi` or `--fi`: Whether to use respectively best improvement or first improvement. Maximally one of these parameters may be provided.

It is also possible to execute all the possible configurations (which constructive method, redundancy elimination or not and possibly an improvement method) from exercise *1.1* and *1.2* on all the instances using a shell file called `run.sh`. These file should be called like this:

```
run.sh PATH/TO/a.out PATH/TO/INSTANCES/DIRECTORY PATH/TO/RESULTS/DIRECTORY
```

These are all relative paths. This file works as follows: for each configuration, the binary file is run with that configuration on each instance. Each execution gives us a cost of the resulting solution. For each instance, this cost is written to a file that describes the configuration, e.g. `ch1+re+bi.txt`. After a configuration is run on all the instances, the time that it took to be executed on all instances is written to a file with the time of each configuration in an exercise. These files are for exercise *1.1* and *1.2* respectively `ex11_durations.txt` and `ex12_durations.txt`. 20 is always used as a seed to conduct the experiments.

When software to execute *R* code is installed, the file `analysis.R` can be executed to analyze the results obtained by running the `run.sh` file. In the

command line, this can be called like this:

```
Rscript analysis.R PATH/TO/best-known.txt PATH/TO/RESULTS/DIRECTORY
```

The results of this analysis are then printed to the command line.

4. Results

4.1. Exercise 1.1

We first conduct experiments using the code with all constructive heuristics, both with and without redundancy elimination. Statistics about the average % deviation from the best results, total computation time and the improvement of the results with redundancy elimination in relation to those without can be seen in the following table:

	Avg. % deviation	Total computation time (seconds)	% that profit from RE	min improvement	avg. improvement	max. improvement
ch1	3507.483	2				
ch1+re	2704.963	3	100%	65	899.065	1962
ch2	46.357	8				
ch2+re	10.941	8	100%	1	63.661	222
ch3	9.868	7				
ch3+re	13.578	7	100%	1	65.355	230
ch4	13.5093	6				
ch4+re	7.675	6	91.935%	0	12.806	82

We see that, in most cases, the resulting cost of a solution of an instance is better when redundancy elimination is applied afterwards. The improvement is the highest when we use *CH1*, but we can see that the average percentage deviation from the costs of the best solutions is still quite high. Redundancy elimination can't fully reduce the effects of the bad choices of sets. *CH4* is the best one in choosing sets, as the average deviation is the lowest when we don't use redundancy elimination. As a result, redundancy elimination can't even always improve the result, which can be seen in the table. The average improvement for *CH4* is also the lowest.

To use a t-test, we need to have a normal distribution. To check this, we used the *Shapiro* test with the null hypothesis that the data are normally distributed. Because the p-value of this test for either the CHx or CHx+RE configuration was lower than 0.05, we determine with a significance level of 0.05 the null hypothesis can be rejected and that there is not a normal distribution for that configuration. As a result, we use the *Wilcoxon Rank-sum* test instead. P-values for *Wilcoxon Ranks-sum* tests to see if there is a difference between the solution quality of a result obtained by a constructive algorithm and the same algorithm with redundancy elimination applied afterwards are shown in the following list:

- ch1 and ch1+re: 7.766e-12
- ch2 and ch2+re: 7.763e-12
- ch3 and ch3+re: 7.764e-12

- ch4 and ch4+re: 5.283e-11

It can be seen that there is always a significant difference between the result of using a constructive algorithm and the result of the use of the same algorithm with redundancy elimination applied afterwards.

4.2. Exercise 1.2

We now conduct experiments using *CH1* and *CH4* with or without redundancy elimination and with best improvement or first improvement. Statistics about the average % deviation, total computation time and percentage of results that profit from the local search phase can be seen in the following table:

	Avg. % deviation	**Total computation time (seconds)**	**% that profit from local search**
ch1+bi	11.165	370	100%
ch1+fi	10.947	16	100%
ch1+re+bi	11.117	336	100%
ch1+re+fi	10.943	17	100%
ch4+bi	7.011	21	91.935%
ch4+fi	6.990	15	91.935%
ch4+re+bi	7.071	15	46.774%
ch4+re+fi	7.071	13	46.774%

We see here that the local search phase has a big effect when we use it on results obtained using *CH1*, because the average % deviation decreases a lot and every result profits from it. Because of the fact that a lot of improvement has to be done and the inner workings of best improvement, we have a big total computation time when applying this local search algorithm on *CH1* results. Because *CH4* on its own already results in quite good costs, it cannot always be improvement anymore by the local search phase. This is even more the case when we first apply redundancy elimination.

In the following table, p-values of the *Wilcoxon Rank-sum* test between 2 different configurations can be seen. Because this test is communicative (the order of the configurations in the function call doesn't matter), we only show the values above the diagonal of the table. Green values signify that the difference between the 2 results is significant on a significance level of 0.05. Otherwise, values are colored red.

/	ch1+re+fi	ch1+re	ch1+re+bi	ch1+fi	ch1	ch1+bi	ch4+re+fi	ch4+re	ch4+re+bi	ch4+fi	ch4	ch4+bi
ch1+re+fi		7.77e-12	0.338	0.683	7.77e-12	0.258	5.68e-08	1.63e-06	5.68e-08	2.47e-08	0.000159	2.59e-08
ch1+re			7.77e-12	7.77e-12	7.77e-12	7.77e-12	7.77e-12	7.77e-12	7.77e-12	7.77e-12	7.77e-12	7.77e-12
ch1+re+bi				0.83	7.77e-12	0.554	1.47e-08	3.95e-07	1.47e-08	7.64e-09	0.000206	7.64e-09
ch1+fi					7.77e-12	0.225	2.71e-08	7.47e-07	2.71e-08	1.16e-08	0.00012	1.16e-08
ch1						7.77e-12	7.76e-12	7.76e-12	7.76e-12	7.76e-12	7.76e-12	7.76e-12
ch1+bi							1.01e-08	2.62e-07	1.01e-08	4.98e-09	0.000221	4.98e-09
ch4+re+fi								2.7e-06	NaN	0.281	5.28e-11	0.361
ch4+re									2.7e-06	4.04e-06	5.28e-11	4.42e-06
ch4+re+bi										0.281	5.28e-11	0.361
ch4+fi											5.28e-11	1
ch4												5.28e-11
ch4+bi												

It can be seen that there is never a significant difference between the results of a constructive algorithm with or without redundancy and with local search (first improvement or best improvement) applied afterwards. This means that, although both configurations and so the inner workings may differ, the resulting costs are much alike.