Implementation Exercise 1
Heuristic Optimization

Jens Nevens - 500093
jens.nevens@vub.ac.be
Academic Year 2015 - 2016
Vrije Universiteit Brussel

April 4, 2015

1 Exercise 1

This section describes the first part of the implementation exercise. This covers four constructive heuristics and redundancy elimination. Section 1.1 covers the implementation of the different heuristics. Section 1.2 will describe the experimental results, followed by an interpretation and discussion of these results in Section 1.3.

Before starting with the implementation, first a note on semantics. In this implementation of the Set Covering Problem, the elements to be covered are called rows, while the subsets responsible for covering them are called columns. Throughout this report, both terms will be used.

1.1 Implementation

Instance Representation: In order to efficiently find information about the current instance, several parameters are used to represent the instance internally. First of all, there are the parameters `m` and `n` which represent the number of rows and columns respectively. Next, there are the arrays `nrow` and `ncol`. The i'th element of each of these arrays contains the number of rows covered by column i for `nrow` and the number of columns covering row i for `ncol`. Another pair of variables is `row` and `col`. These both are arrays of arrays. Each i'th element of the `row` array contains the rows that are covered by the i'th column, while each i'th element of the `col` array contains the columns that cover the i'th element. The final pair of variables is the `cost` and `ccost` arrays. The former contains the cost for each column, whilst the latter contains the cover-cost for each column, i.e. the cost of the column divided by the number of elements covered by that column.

Solution Representation: Next to efficiently representing the instance at hand, it is also necessary to have appropriate data structures to represent the (partial) solutions. This is done by the `Solution` struct. This C struct contains seven variables, the first of which are `x` and `y`. These are binary arrays with respectively the length of the number of columns and number of rows. A 1 at position i indicates that the i'th column or row is selected for the current (partial) solution. The variable `fx` contains the cost of the current (partial) solution. The array `ncol_cover` contains the number of columns covering element i. Likewise, the array of arrays `col_cover` contains the identities of these columns. Finally, the variables `un_rows` and `un_cols` contain the number of un-covered rows and un-used columns respectively.

Reading an Instance: Several methods are responsible for reading an instance and instantiating the appropriate data structures. These will be briefly discussed here.

`readParameters` This method parses the command line arguments provided to the program. These are the seed for the random number generator, a path to the instance file, a path to the output file, which constructive heuristic to run and 3 flags indicating whether or not to perform redundancy elimination, iterative best improvement or iterative first improvement.

`readSCP` This method reads the instance file, allocates memory and instantiates the data structures discussed in the first paragraph.

`initialize` Finally, this method allocates memory for the `Solution` struct discussed in the previous paragraph. It also makes sure the variables of the struct are correctly instantiated.

General methods: Several methods implement operations on the (partial) solution that can be used by all heuristics. These will be discussed here.

`addSet` The `addSet` method adds a certain set to a given partial solution. Let's call this set s. This requires performing an update on each element of the partial solution struct. To start, one needs to indicate that set s is used in the partial solution. This is done by flipping the bit on position s in the binary array `x`. Next, the cost of the partial solution `fx` is incremented with the cost of set s. Also, the number of un-used sets is decremented. After that, a number of operations needs to be performed for each element

covered by set s. These elements are denoted by e. For each of these elements e, the number of columns covering the element is incremented. Also, if element e was not yet present in the partial solution, this needs to be indicated. This is done by flipping the bit in the binary array `y`. Important to note is that when this bit is actually flipped from 0 to 1, then also the number of un-covered elements needs to be decremented. Finally, set s needs to be added to the array of sets covering element e, again for every element e.

`removeSet` Similarly to adding a set does removing a set also require several bookkeeping operations on the partial solution. These are basically the reverse operations of those specified in the previous paragraph. When a set s is removed from a partial solution, one needs to indicate in the binary array `x` that this set is no longer selected. Next, the partial cost `fx` needs to be decremented with the cost of set s. Also, the number of un-used columns is incremented. Up to three operations need to be performed for every element e covered by set s. First of all, the set itself needs to be removed from the array of sets covering element e. This is done by altering the `col_cover` array on position e. The `shift` helper method makes sure that any sets occurring after s in this array are shifted to the left. Next, the number of columns covering element e is decremented. When this has as a consequence that element e is no longer covered by any sets, the number of un-covered elements is incremented and bit e in the binary array `y` is flipped.

`redundant` Before adding a set to a solution, it is necessary to check whether or not that set is redundant. For a set s, this is done by going over all elements e covered by that set. As soon as one of those elements e is not yet covered, i.e. the binary array `y` indicates a zero, the set s is not redundant and can be added.

`isBetter` Another required operation is determining whether a given set s is better or worse than another set s'. Since this needs to be checked frequently, for different pairs of sets, it is abstracted into the `isBetter` method. This method will first determine the best set based on the cost, i.e. the one with the lowest cost. In case the costs are equal, the number of elements covered by each set is considered. The set covering the most elements is chosen. If this also results in a tie, one of the sets is chosen at random.

`isSolution` To check whether a given solution struct is a valid solution

to the Set Covering Problem, the `un_rows` parameter of the struct is used. When this is zero, no elements remain uncovered and thus the SCP is satisfied.

Random Construction: The random constructive heuristic is implemented in two steps. First, a random un-covered element is chosen. In order to check if an element is un-covered, the binary array `y` of the solution representation is used. Next, a random un-used set covering the chosen element is picked. Similar to the elements, in order to check if a set is un-used, the binary array `x` is used. Finally, the `addSet` method, explained above, is used to add the chosen set to the partial solution. This is repeated until the `isSolution` condition is satisfied. One should not that the `redundant` method, described above, is not used to make sure the set is not redundant. Indeed, this is not necessary, since the chosen element is un-covered and thus any set covering this element is by definition not redundant. The only thing that remains unexplained is how a random element or random set is chosen. The `pickRandom` method is responsible for this. Given a range of N integers in which the returned number should occur, this method creates N equal size buckets. Then, a loop generates random numbers until one of them falls within a bucket and this bucket is returned. This way, all buckets are equally likely, i.e. the random number is generated from a uniform distribution.

Cost-based Construction: Several cost-based constructive heuristics are implemented using only three methods. All three of these heuristics (static cost-based, static cover cost-based and adaptive cover cost-based) will be explained in this paragraph. They all work according to the same principle, namely choosing the set with the smallest cost and adding it to the solution if it is not already added and it's not redundant. They only vary in how the cost for a set is determined. The main entry point for these heuristics is the `costBased` method. This method will loop over all sets, get the cost for each of them and retain the one with the smallest cost. At the end, this set is added to the partial solution. This basic loop is executed until the `isSolution` condition is satisfied. In order to support all three heuristics, a separate method is used to get the cost of a given set, namely the `getCost` method. Depending on the heuristic selected, this method will either return the cost as given in the instance file, the cost divided by the number of elements covered by that set or the cost divided by the number of elements that would

be additionally covered if the given set were added. The static cover cost is computed once during initialization, whilst the computation of the adaptive cover cost is postponed until necessary. This is done by the `adaptiveCost` method. For a given set s, this method will go through all elements covered by s and increment a counter when these are not covered. At the end, the cost of set s is divided by the result of this counter.

Redundancy Elimination: After a solution has been constructed by a random or cost-based constructive heuristic, it is possible that some sets in this solution are redundant. A set s is considered redundant when all elements e covered by this set are covered by one or more other sets. Removing this set would have no negative effect on the final solution, since all elements covered by this set remain covered by other sets. The `eliminate` method removes these redundant sets and does this in such a way to minimize the cost. In order to do this, the build-in quick-sort is used to sort all sets in a descending fashion based on their cost. Afterwards, a loop goes through all sets s. A set is only considered, in this loop, when it is part of the current solution. Then, all elements e covered by this set are considered. As soon as one element covered by this set has only one set covering it, the set is no longer eligible for removal, since removing it would break the solution. If the end of the loop over all elements e is reached, one can conclude that the set is redundant and it can thus be safely removed. This process is repeated until no more sets can be removed.

1.2 Experimental Results

Several results have been obtained by running the heuristics described above. In order to assure uniformity across experiments, the random number generator is seeded with the same number, in this case the number 1234567. This section describes different statistics obtained from the heuristics. An interpretation and discussion of these results is provided in Section 1.3. All statistics mentioned here are computed by R code.

A first important result is the average percentage deviation from the best known solution. This is described in Table 1. Secondly, there is the computation time. Table 2 contains the computation time for each heuristic over all instances. This is indicated in number of seconds.

Table 3 reports the fraction of instances that benefit from applying redundancy elimination, i.e. the fraction for which a better cost solution was

Algorithm	Deviation
CH1	3587.35
CH1+RE	2722.21
CH2	47.23
CH2+RE	10.94
CH3	50.19
CH3+RE	13.85
CH4	12.70
CH4+RE	7.53

Table 1: Average Percentage Deviation

Algorithm	Time
CH1	2
CH1+RE	2
CH2	3
CH2+RE	3
CH3	3
CH3+RE	2
CH4	5
CH4+RE	5

Table 2: Total Computation Time

found.

Algorithm	Fraction
CH1+RE	1
CH2+RE	1
CH3+RE	1
CH4+RE	0.90

Table 3: Benefiting Instances

Next, Table 4 describes statistics about the improvement obtained by applying redundancy elimination. This table reports the minimum, maximum and mean reduction in percentage deviation from the optimal solution obtained by redundancy elimination.

Algorithm	Minimum	Maximum	Mean
CH1+RE	103.52	2592.86	865.15
CH2+RE	13.33	57.14	36.28
CH3+RE	13.33	57.14	36.34
CH4+RE	0	12.79	5.16

Table 4: Improvement by Redundancy Elimination

Finally, Tables 5 and 6 describe the obtained p-values after executing a paired t-test on the percentage deviation from the best known solution. The significance level for this t-test is $\alpha = 0.05$. A full matrix, comparing each

algorithm to any other algorithm, can be found in the file `ttest.txt`.

	CH2	CH3	CH4
CH1	5.57×10^{-16}	6.01×10^{-16}	4.06×10^{-16}
CH2		1.19×10^{-3}	1.10×10^{-36}
CH3			5.29×10^{-38}

Table 5: p-values of paired t-test

Algorithm	Algorithm	p-value
CH1	CH1+RE	6.82×10^{-15}
CH2	CH2+RE	3.16×10^{-40}
CH3	CH3+RE	1.08×10^{-39}
CH4	CH4+RE	6.37×10^{-20}

Table 6: p-values of paired t-test

1.3 Discussion

In general, Table 1 gives a fairly good overview over the different constructive heuristics. Several conclusions can be drawn from this table. First of all, it is clear that the random constructive heuristic performs the worst, while the adaptive cover cost-based heuristic is the best. The static cost-based heuristic and static cover cost-based heuristic perform approximately identical, i.e. the solutions they generate are equally far away from their respective optima. From Table 2 it is also clear that almost all of the constructive heuristics take equally long to compute. The exception is the adaptive cover cost-based heuristic, which only takes a few seconds more. The reason for this is fairly simple, namely the fact that the cost has to be recomputed for a set when necessary.

One also notes that redundancy elimination is a successful measure to reduce the cost of a solution. From Table 1 it is clear that the solution quality after applying redundancy elimination has improved. The percentage deviation for each algorithm after applying redundancy elimination is smaller.

The benefit of redundancy elimination can also be seen from Table 3. For all but the adaptive cover cost-based heuristic does redundancy elimination reduce the cost of the solution for all instances. Nevertheless, 90% of the

instances also benefits from redundancy elimination when using the adaptive cover cost-based heuristic. This is an indicator that the quality of the initial solutions generated by this heuristic is quite good. When looking at Table 2, it is clear that redundancy elimination does not require a significant amount of additional time.

The same trend is noticeable when studying Table 4. The redundancy elimination algorithm achieves the greatest improvement when applied after the random constructive heuristic, i.e. on average a decrease in percentage deviation of approx. 860%. This is a straightforward result since the solutions found by this heuristic are poor, i.e. there is a large margin for improvement. Nevertheless, the quality of these solutions remains poor, simply because of the constructive method. Again, the static cost-based heuristic and static cover cost-based heuristic behave approximately equal. The redundancy elimination procedure is able to improve the solution quality by an equal amount for these two heuristics. The smallest improvement is noted for the adaptive cover cost-based heuristic. The reason for this being the quality of the initial solutions which is quite good. On average, a decrease in percentage deviation of approx. 5% is noted.

Finally, one notes from Tables 5 and 6 that the differences between the results of all algorithms, with and without redundancy elimination, are significant. Indeed, the p-value is smaller than 0.05 for each pair of algorithms. As was mentioned, the static cost-based and static cover cost-based heuristic share some similarities. Nevertheless, the results they obtain remain significantly different.

2 Exercise 2

This section describes the second part of the implementation exercise. This includes both the iterative best- and first-improvement algorithms. The implementation of these algorithms is described in Section 2.1. Section 2.2 describes different statistics about these algorithms, while Section 2.3 interprets and discussed these results.

2.1 Implementation

Solution Copy: As was mentioned in Section 1.1, the `Solution` struct is used to represent a solution to the SCP. In order to support the iterative best- and first-improvement algorithms, a second instance of this struct is created, called `cpy`. This second instance is used as a working copy of the solution. The iterative improvement algorithms

can alter this solution and, when an improvement has been achieved, the solution is copied over to the true solution struct. This is done by the `copySolution` method, which does a deep copy of one struct to the other. There is also the `initCopy` method, which initializes the `cpy` struct in the same manner as the `initialize` method described earlier.

First Improvement: On a high level, the iterative first improvement algorithm works as follows. It receives the initial solution as input and tries to improve it. This is done by sequentially going through the neighbourhood of the current solution. As soon as a better neighbour is found, this neighbour is taken to be the new current solution. This process is repeated until no more improvement is achieved, i.e. until convergence into a local minimum. In the Set Covering Problem, a neighbour is defined in terms of sets. A neighbour is created by first removing a set present in the current solution and then adding other sets to re-complete the solution. Removing a set is done by the `removeSet` method, described in Section 1.1. Re-completing the solution is done by a separate method, called `replaceSet`. This method will add sets, different from the removed one, until the `isSolution` predicate is satisfied. Also, this method will add the lowest cost sets first and will only consider the adaptive cover cost for this. These operations are performed on the `cpy` struct. The reason for this being the fact that when a neighbour turns out to be worse, the original (and better) solution is not lost, since it is stored in another struct. When the neighbour is in fact a better solution, the contents of the `cpy` struct are copied over to the true solution. The methods used for this are described in the previous paragraph. When redundancy elimination is active, it is applied each time to the newly found solution. As mentioned, this entire process is repeated until no more improvement is achieved.

Best Struct: An additional struct is used for the iterative best improvement algorithm. The so called `best` struct keeps track of 4 parameters: the set removed from the solution, the sets added to the solution, the number of sets added to the solution and the cost of the solution. This struct can be initialized by the `initBest` method.

Best Improvement: The iterative best improvement algorithm is quite similar to the first improvement algorithm. It also receives the initial solution as input and tries to improve it. However, in contrast to the first improvement method, the best improvement method considers

all neighbours in the neighbourhood. Afterwards, the best of those is chosen as the new current solution. For this, both the `cpy` struct and `best` struct are used. First of all, the same methods as for the first improvement are used to remove a set and complete it with other sets. However, for the best improvement algorithm, the sets which are added are kept in an array. If it turns out the neighbour (in the `cpy` struct) is better than the current solution, the `best` struct is used. This struct keeps track of the removed set, the sets added and the cost of the solution. After all neighbours in a neighbourhood have been considered, the `best` struct contains which sets to add and remove in order to obtain the best neighbour. The `applyBest` method is responsible for applying these changes to the original solution. Afterwards, the `best` struct is reinitialized, the best neighbour is copied over to the `cpy` struct for the next round and redundancy elimination is applied, if necessary. This entire process is again repeated until no more improvement is achieved.

2.2 Experimental Results

As was the case in the first part of the implementation exercise, running the algorithms described above provides some meaningful data in order to compare their performance. These different statistics will be described here. An extensive discussion of these statistics follows in Section 2.3. In order to maintain conformity, the same number was used to seed the random number generator. This was the number 1234567. The different statistics described below are obtained through R code.

These tables are roughly the same as described in Section 1.2. The first result, seen in Table 7, shows the average percentage deviation from the best known solution. This is followed by the computation time over all the instances, shown in Table 8.

Algorithm	Deviation
CH1+FI	11.06
CH1+RE+FI	11.01
CH1+BI	11.19
CH1+RE+BI	10.66
CH4+FI	6.77
CH4+RE+FI	6.83
CH4+BI	6.76
CH4+RE+BI	6.74

Table 7: Percentage Deviation

Algorithm	Time
CH1+FI	15
CH1+RE+FI	17
CH1+BI	394
CH1+RE+BI	228
CH4+FI	14
CH4+RE+FI	11
CH4+BI	33
CH4+RE+BI	11

Table 8: Total Computation Time

The fraction of instances that benefits from the additional iterative improvement step can be seen in Table 9. The algorithms in Table 9a are compared against the regular algorithms, i.e. algorithms without redundancy elimination. Table 9b contains the algorithms which are compared against the algorithms with redundancy elimination.

Algorithm	Fraction
CH1+FI	1
CH1+BI	1
CH4+FI	0.90
CH4+BI	0.90

(a) Compared to regular algorithm

Algorithm	Fraction
CH1+RE+FI	1
CH1+RE+BI	1
CH4+RE+FI	0.48
CH4+RE+BI	0.48

(b) Compared to algorithm with redundancy elimination

Table 9: Benefiting Instances

Finally, Tables 10 and 11 show the p-values after doing a paired t-test on the percentage deviation from the best known solution. Again, the significance level for these tests is $\alpha = 0.05$. The file `ttest.txt` contains the p-values for all pairs of algorithms.

2.3 Discussion

The statistics described in Section 2.2 allow to draw several conclusions regarding the iterative improvement algorithms.

First of all, when looking at Table 7, it is clear that the adaptive cover cost-

	CH1+BI	CH4+FI	CH4+BI
CH1+FI	0.78	1.31×10^{-08}	1.18×10^{-08}
CH1+BI		3.31×10^{-11}	2.69×10^{-11}
CH4+FI			0.16

Table 10: p-values of paired t-test

Algorithm	Algorithm	p-value
CH1+FI	CH1+RE+FI	0.91
CH1+BI	CH1+RE+BI	0.30
CH4+FI	CH4+RE+FI	0.32
CH4+BI	CH4+RE+BI	0.80

Table 11: p-values of paired t-test

based heuristic, together with iterative improvement, is the best algorithm overall. It outperforms any other algorithm described in this paper thus far. One also notes that the percentage deviation for this heuristic, without redundancy elimination, is the same for the first- and best-improvement algorithms. These algorithms probably find solutions in the same local minima. On the other hand, when redundancy elimination is activated, one notices that the best-improvement algorithm gets closer to the optimal solution, while the first-improvement algorithm gets further away from it. An explanation for this phenomenon might be that the combination of redundancy elimination and first-improvement leads to a worse local minimum, since not all neighbours are considered. At the same time, the combination of redundancy elimination and best-improvement leads to a slightly better local minimum. This is possible since redundancy elimination is applied after each iterative improvement step. Thus, it is possible that the solution is nudged towards a slightly better or slightly worse local minimum by this procedure.

Another noteworthy remark can be made from the numbers in Table 7. Namely, one sees that the iterative improvement algorithm inflicts a huge improvement for the random constructive heuristic. This is even clearer when comparing these percentage deviations to the ones in Table 1. When studying Table 8, one sees that this huge improvement is achieved quite fast for the first-improvement algorithm, but very slow for the best-improvement algorithm. A plausible cause for this might be the following; one already knowns that

the initial solution generated by the random constructive heuristic is poor. One also knows that the first-improvement algorithm moves on to the next iteration as soon as an improving neighbour has been found. But, since the initial solution is so poor, each time it finds an improving neighbour very fast. On the other hand, the best-improvement algorithm has to consider each neighbour individually. So, a lot of extra work has to be done for an iteration to be completed. And, since the initial solution is poor, a lot of iterations are needed to get to a local minimum. Additionally, the best-improvement algorithm is more heavy from a computational view. More memory and operations are needed for a single iteration. On a final note, one does also see that interleaving the best-improvement algorithm with redundancy elimination again reduces the time by quite a margin.

When looking at the second half of Table 8, one sees that the first-improvement algorithm after the adaptive cover cost-based heuristic is executed quite fast, with or without redundancy elimination, and again the best-improvement algorithm is slower. However, not nearly as slow as was the case for the random constructive heuristic. Also, interleaving the iterative improvement steps with redundancy elimination speeds up the converge to a local minima.

Table 9a shows that both iterative improvement algorithms are beneficial. This is especially the case when they are applied after the random constructive heuristic. All instances benefit from the additional improvement phase. This is to be expected, since the solutions generated by this heuristic are poor. The fraction of benefiting instances drops below 1 when being applied after the adaptive cover cost-based heuristic. When the redundancy elimination procedure is turned off, 90% of the instances benefit from iterative improvement. This is a logical consequence, since the solutions generated by this heuristic alone are already of a good quality.

The same trend is noticeable from Table 9b. All instances benefit from the additional iterative improvement phase, for both iterative improvement algorithms, after the random constructive heuristic with redundancy elimination. The results are not as good for the adaptive cover cost-based heuristic. Both iterative improvement algorithms deliver a benefit of less than 50% for this heuristic, compared to the original heuristic with redundancy elimination. An explanation for this is the fact that this heuristic, together with redundancy elimination, already results in solutions that are a local minima for several instances. It is thus impossible for the iterative improvement algorithms to find a better solution.

Finally, Tables 10 and 11 contain the p-values obtained after executing a paired t-test. From Table 10, one notices that the first- and best-improvement

algorithms return results which are not significantly different when applied after the same constructive heuristic. This can be seen from the p-value of the random constructive heuristic (0.78) and of the adaptive cover cost-based heuristic (0.16). Also, from Table 11, it can be concluded that the redundancy elimination procedure has no big impact on the outcome of the algorithms. Indeed, this can be seen since for both heuristics and for both the best- and first-improvement algorithms, the results are not significantly different.

3 Running the Code

Command line arguments are used to set different parameters and select the appropriate algorithms to run. These are described in Table 12.

Argument	Type	Effect
`--seed`	Mandatory	Sets the seed for the random number generator
`--instance`	Mandatory	Specifies which instance to use
`--output`	Mandatory	Specifies where to output data
`--ch1`	Flag	Selects the random constructive heuristic
`--ch2`	Flag	Selects the static cost-based heuristic
`--ch3`	Flag	Selects the static cover cost-based heuristic
`--ch4`	Flag	Selects the adaptive cover cost-based heuristic
`--re`	Flag	Selects the redundancy elimination procedure
`--fi`	Flag	Selects the iterative first improvement algorithm
`--bi`	Flag	Selects the iterative best improvement algorithm

Table 12: Command Line Arguments

In order to run the program correctly, the `./lsscp.c` file has to be executed with a seed, an instance, an output file and one of the heuristics. The redundancy elimination flag and/or one of the iterative improvement flags can be added as well.

To facilitate running multiple experiments in batch, the `run.sh` file is present. This file will run all possible configurations for all instances, write the output to separate files and keep track of how long each algorithm runs. This file can be executed by the following command: `./run.sh ./path/to/executable ./path/to/instances ./path/to/output-folder`. The output folder will contain a file for each configuration, with the cost found by that configuration for each instance. It will also contain two files indicating the time each configuration took to complete all instances.

Additional output can be generated by executing the two R files: `statistics.R` and `t-tests.R`. These files compute the information described in Sections 1.2 and 2.2. Additional files will be created in the output folder.