Report implementation exercise 2
Heuristic Optimization

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

1 Implementation

In this implementation exercise, 2 different algorithms had to be implemented. First the Iterated Local Search algorithm will be explained, followed by an explanation of Ant Colony Optimization. Both algorithms run until a certain termination criterion is met. The algorithm calls before each iteration a function to know if it should stop. Depending on the experiment, this termination criterion is based on a time and/or cost threshold. Each time the algorithm finds a new best solution, a function named `notify_improvement` is called. When the path to a trace file is passed to the program, the function writes the time when this solution was found and the cost of this solution to that file.

1.1 Iterated Local Search

This implementation was based on the paper by Jacobs and Brusco (1995). We will first explain the global framework of the algorithm, after which the local search phase is explained.

1.1.1 Global framework

As mentioned, the algorithm keeps iterating as long as a termination criterion is not met. To make the explanation easier, we will first look at pseudo code, shown in Algorithm 1.

Algorithm 1: Global framework of Iterated Local Search

```
start_solution ← construct_solution_using_CH4()
redundancy_elimination(start_solution)
overall_best_solution ← start_solution
current_best_solution ← start_solution
while termination criterion is not met do
    for TL times do
        new_solution ← copy(current_best_solution)
        local_search_phase(new_solution)
        delta ← cost(new_solution) − cost(current_best_solution)
        if delta ≤ 0 then
            current_best_solution ← new_solution
            if cost(new_solution) < cost(overall_best_solution) then
                overall_best_solution ← new_solution
            end
        else if random_with_probability(e^{−delta/T}) then
            current_best_solution ← new_solution
        end
    end
    T = T ∗ CF
end
return overall_best_solution
```

We start by constructing a complete solution and applying redundancy elimination (explained in implementation exercise 1) to it. While in the paper *CH2* (static cost-based greedy values) is used to build an initial solution, I used *CH4* (adaptive cover cost-based greedy values) instead because this construction heuristic resulted in lower costs on most instances in implementation exercise 1.

In the global framework itself, it can be seen that there is an outer loop which is executed as long as the termination criterion is not met. There is also an inner loop which is always executed `TL` (temperature length) times. In each inner loop, we copy the currently best solution and apply the local search phase, explained in Section 1.1.2.

If the new solution is better than the currently best one, we always set the new one as the current best. We also set the overall best one to the new solution if the cost of the new one is strictly lower. This is a deviation from the paper, where they stated that they would update the overall best whenever the current best improved. This might give unwanted consequences as the new solution may not be the best found in the overall execution of

the algorithm, because the current best (the one that is improved in this iteration) may have been obtained by choosing it with a probability (which is explained in the next paragraph). As explained, my algorithm does not have this consequence.

When the new solution isn't better, we only keep the new one as the current solution with a certain probability. In the algorithm `T` is called the temperature and determines together with `delta` the probability of keeping a solution that is worse than the current best. This probability is $e^{-\delta/T}$. Because δ is always positive in this case and the temperature is positive as well, we get a negative exponent. If δ is high, the new solution is a lot worse and the chance of keeping it is low. The opposite is true for small differences. A higher temperature means that the probability of keeping the worse solution is also higher.

After `TL` times, the temperature is reduced by a factor of `CF` (cooling factor). This means that every `TL` times, the chance of keeping a worse solution is relatively lower due to a lower temperature.

1.1.2 Local search phase

In this step, parts of the solutions are removed and the solution is rebuilt. We will again take a look at the pseudo code, which is shown in Algorithm 2.

Algorithm 2: Local search phase of Iterated Local Search

Input: *solution* on which to apply local search
Output: *solution* with local search applied

```
1  D ← ⌈used_sets(solution) * ρ1⌉
2  E ← ⌈max_set_cost(solution) * ρ2⌉
3  randomly_remove_sets(D, solution)
4  while uncovered_elements(solution) do
5  |  candidate_sets ← unused_sets_below_threshold(E, solution)
6  |  alfa ← ∅
7  |  for set ∈ candidate_sets do
8  |  |  alfa[set] = cost(set)/added_elements(solution, set)
9  |  end
10 |  min_alfa ← minimum(alfa)
11 |  for set ∈ candidate_sets do
12 |  |  if alfa[set] ≠ min_alfa then
13 |  |  |  alfa ← alfa \ {set}
14 |  |  end
15 |  end
16 |  chosen_set ← random_from(candidate_sets)
17 |  add_to_solution(solution, chosen_set)
18 end
19 redundancy_elimination(solution)
20 return solution
```

Here, ρ_1 ($0 < \rho_1 \leq 1$) and ρ_2 ($0 < \rho_2$) are parameters passed to the program on beforehand and don't change anymore. ρ_1 is the percentage of sets that have to be removed (randomly) from the solution, which results in $\lceil used\_sets_{solution} * \rho_1 \rceil$. The value ρ_2 is used to limit which sets can be used in the rebuilt solution. This value is preferably larger than 1 to allow sets with a higher cost but with more covering elements or to have temporarily worse solutions to be able to overcome local minima. Using this value, we get $\lceil max\_set\_cost_{solution} * \rho_2 \rceil$ for `E`, where $max\_set\_cost_{solution}$ is the cost of the set with the highest cost in the solution. After randomly removing `D` sets, we keep iterating until all elements are covered. Each time, we first make a set of candidate sets which aren't used in the solution yet and which have a cost lower or equal than `E`. Afterwards, we keep for each candidate set an alfa value, which is the cost of the candidate set divided by the number of elements that aren't covered yet but that would be covered if we would add the set to the solution. The latter is in the actual code obtained using an array reference, as to avoid needlessly computing it again. Its values are

updated when a set is added or removed from a solution.

Then we remove sets of which this alfa value is not minimal. Afterwards, we randomly choose a set that is still a candidate and add it to the solution.

When all elements are covered, we apply redundancy elimination to the solution.

1.1.3 Parameter values

To execute the algorithm I used the *PS3* parameter values from the original paper. These are: $T = 1.3$, $TL = 100$, $CF = 0.9$, $\rho_1 = 0.4$ and $\rho_2 = 1.1$. The usage of other parameter settings from the paper or my own choices of parameter values didn't lead to better results.

1.2 Ant Colony Optimization

My implementation of the Ant Colony Optimization algorithm was based on the paper by Ren et al. (2010). First, the global framework is explained, after which details of the ant construction and local search phase will be given.

1.2.1 Global framework

We first give pseudo code of the global framework, after which some parts are explained in more detail. This pseudo code can be seen in Algorithm 3.

Algorithm 3: Global framework of Ant Colony Optimization

```
1  τ_max ← 1/((1 − ρ) ∗ cost(all_sets))
2  τ_min ← ϵ ∗ τ_max
3  pheromone_trails ← list()
4  for set ∈ all_sets do
5  |  pheromone_trails[set] ← random_between(τ_min, τ_max)
6  end
7  while termination criterion is not met do
8  |  for i ← 1 to n_a do
9  |  |  ant ← construct_ant(pheromone_trails)
10 |  |  apply_local_search(ant)
11 |  |  if no global_best or cost(ant ≤ cost(global_best) then
12 |  |  |  global_best ← ant
13 |  |  |  τ_max ← 1/((1 − ρ) ∗ cost(global_best))
14 |  |  |  τ_min ← ϵ ∗ τ_max
15 |  |  end
16 |  end
17 |  Δτ ← 1/cost(global_best)
18 |  for set in global_best do
19 |  |  pheromone_trails[set] ← ρ ∗ pheromone_trails[set] + Δτ
20 |  |  if pheromone_trails[set] > τ_max then
21 |  |  |  pheromone_trails[set] ← τ_max
22 |  |  else if pheromone_trails[set] < τ_min then
23 |  |  |  pheromone_trails[set] ← τ_min
24 |  |  end
25 |  end
26 end
```

The algorithm starts by calculating τ_{max} and τ_{min}. At first, τ_{max} uses the worst case scenario, in which all sets would be used in the solution. τ_{min} is just a fraction of τ_{max}, determined by ϵ. This results in the following formulas:

$$\tau_{max} = \frac{1}{(1-\rho) * \sum_{i=0}^{n} c_i} \tag{1}$$

$$\tau_{min} = \epsilon * \tau_{max} \tag{2}$$

A pheromone value (τ_j) is then instantiated for each set. This represents the desirability of a set to be selected in the construction of a solution. Initially, the value for each set is set to a random value between τ_{min} and τ_{max}.

Just like the previous algorithm, this one keeps iterating until the termination criterion is met. In each iteration, for each ant (in total n_a) a solution is built and local search is applied to it. If the resulting solution is better than the global best one, we set the new solution as the global best and we update τ_{max} and τ_{min}. Instead of the total cost of all sets, here we use the costs of the sets in the new global best solution to calculate τ_{max}.

After each ant was used to build solutions, we update the pheromone trail of each set. We multiply the old pheromone trail of the set by ρ ($0 < \rho \leq 1$),a parameter which is given on beforehand and does not change. This parameter is usually set to a value less than 1 as to evaporate pheromone trails and as a result to gradually reduce the exploitation of sets. We also increase the pheromone trail by $\Delta\tau = \frac{1}{total\_cost_{global\_best}}$. The value of each pheromone trail τ_j is also restricted to $\tau_{min} \leq \tau_j \leq \tau_{max}$.

The paper on which my algorithm was based also mentions *column inclusion* (adding sets to a solution that uniquely cover an element) and *column domination* (Prohibiting to use a set for which there are sets that cover its elements and have a lower total cost). These did however not lead to better results and only slowed down the execution. Thus, they are not applied.

1.2.2 Construction phase

Just like before, we again give an overview of the phase using pseudo code, after which specifics will be discussed. This pseudo code is shown in Algorithm 4.

Algorithm 4: Construction phase of Ant Colony Optimization

Input: *pheromone_trails*
Output: *ant*: a complete solution

probability_distribution ← *list*()
ant ← *empty_solution*
while *uncovered_elements*(*ant*) **do**
    chosen_element ← *random_of*(*uncovered_elements*(*ant*))
    valid_sets ← *sets_covering*(*chosen_element*) \ *used_sets*(*ant*)
    denominator ← $\sum_{i \in valid\_sets} pheromone\_trails[i] * heuristic\_information(ant, i)^{\beta}$
    for *set* ∈ *all_sets* **do**
        if *set covers chosen_element* ***and*** *set* ∉ *used_sets*(*ant*) **then**
            probability_distribution[*set*] ← *pheromone_trails*[*set*] * *heuristic_information*(*ant*, *set*))$^{\beta}$/*denominator*
        else
            probability_distribution[*set*] ← 0
        end
    end
    chosen_set ← *random_using_pdf*(*probability_distribution*)
    add_to_solution(*ant*, *chosen_set*)
end
return *ant*

We always start with an empty solution (no sets chosen yet) and then continue iterating until all elements have been covered. Each time, we randomly select an element that isn't covered yet in the current solution. Then, we calculate a probability for each unused set that covers that element. This value is equal to:

$$P(j) = \frac{\tau_j * \eta_j^{\beta}}{\sum_{k \in J_i \setminus s} \tau_k * \eta_k^{\beta}} \tag{3}$$

Where $P(j)$ is the probability of the set being chosen, s represents the sets already in the solution and J_i is the collection of all sets covering the chosen element i. η_j (`heuristic_information` in the pseudo code) is the heuristic information of set j. It is equal to the number of elements that would be covered if we would add set j to the solution, divided by the cost of set j. The heuristic information is high when the set is a good candidate, because then the cost would be low and/or it would cover a lot of extra elements. β is a parameter given on beforehand used to amplify or reduce

the effect of the heuristic information. The probability of choosing a set that is already in the solution or don't cover the chosen element is 0. Note that in the actual code, only probabilities for sets covering the element are calculated, instead of probabilities for all the sets. Thus, the algorithm only loops over these sets for calculating probabilities and choosing a set, which saves a lot of computation time.

Calculating this probability for every set, we get a probability distribution. We can then randomly choose a set using this probability distribution and add it to the solution.

1.2.3 Local search phase

To optimize the solution of an ant, we try to remove a set or replace it with with other sets. This can be seen in the pseudo code in Algorithm 5.

Algorithm 5: Local search phase of Ant Colony Optimization

Input: *ant* on which to apply local search
Output: *ant* with local search applied

for $set \in used\_sets\_descending\_cost$ **do**
  $n\_only\_covered\_by\_set \leftarrow 0$
  $only\_covered\_by\_set \leftarrow list()$
  for $element \in elements\_covered\_by\_set(ant, set)$ **do**
    if *element is not covered by other sets in ant* **then**
      $only\_covered\_by\_set \leftarrow append(only\_covered\_by\_set, element)$
      $n\_only\_covered\_by\_set \leftarrow n\_only\_covered\_by\_set + 1$
    end
  end
  if $n\_only\_covered\_by\_set \geq 1$ **then**
    $lowest1 \leftarrow$ *set* with lowest cost in instance covering $only\_covered\_by\_set[0]$
  end
  if $n\_only\_covered\_by\_set = 2$ **then**
    $lowest1 \leftarrow$ *set* with lowest cost in instance covering $only\_covered\_by\_set[1]$
  end
  if $n\_only\_covered\_by\_set = 0$ **then**
    $remove\_set(ant, set)$
  else if $n\_only\_covered\_by\_set = 1$ ***and*** $set \neq lowest1$ **then**
    $remove\_set(ant, set)$
    $add\_set(ant, lowest1)$
  else if $n\_only\_covered\_by\_set = 2$ ***and*** $lowest1 = lowest2$ ***and*** $set \neq lowest1$ **then**
    $remove\_set(ant, set)$
    $add\_set(ant, lowest1)$
  else if $n\_only\_covered\_by\_set = 2$ ***and*** $lowest1 \neq lowest2$ ***and*** $cost(lowest1) + cost(lowest2) \leq cost(set)$ **then**
    $remove\_set(ant, set)$
    $add\_set(ant, lowest1)$
    $add\_set(ant, lowest2)$
  end
end
return *ant*

For each set in the solution of an ant (in descending cost order), we see how many elements that a set covers in the solution, that are only covered by that set in the current solution. In the original paper, this collection of elements for a set j is W_j (`only_covered_by_set` in the pseudo code). If $|W_j| = 0$, then the set isn't the only one covering each element and we can safely remove the set from the solution.

If $|W_j| = 1$, there is one element that is only covered by the set. For this element, we find the set with the lowest cost, covering the element in the instance (`lowest1` in the pseudo code). If this set isn't the one we're trying to replace, we can remove the original set and add this one to the solution.

If $|W_j| = 2$, 2 elements are uniquely covered by the current set in the solution. If, however, the lowest covering set of the first element (`lowest1`) and of the second one (`lowest2`) are the same but aren't equal to the current set, we can again remove the old set from the solution and add the new one (it doesn't matter whether it's `lowest1` or `lowest2` because they are the same).

The last case is when we have again $|W_j| = 2$, but when the 2 lowest covering sets aren't the same. We can however still replace the original set if the total cost of the 2 new sets is lower or equal than the cost of the original set. After removing the old set, we then add the 2 new sets (`lowest1` and `lowest2`) to the solution.

If none of these conditions are satisfied, we just continue with the next set.

1.2.4 Parameter values

To conduct experiments with the Ant Colony Optimization, I used the following parameter values: $\beta = 5.0$, $\rho = 0.99$, $\epsilon = 0.005$ and $n_a = 20$. These are the same parameter values as those in the paper describing the algorithm on which my algorithm was based. I tried different parameter values, but those only gave slightly better results for instances with a low number of elements and sets. The results were worse for bigger instances. Thus, I used the original parameter values.

2 File structure

- `main.c`: Parses arguments passed to the binary, keeps track of the termination criterion, writes to the trace file if necessary and calls the right algorithms depending on the given arguments.

- `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 improvement and best improvement.
- `ils.h` and `ils.c`: Contains functions needed to apply iterated local search.
- `aco.h` and `aco.c`: Consists of the functions necessary to apply ant colony optimization.
- `utils.h` and `utils.c`: Contains help functions unrelated to specific functionalities.
- `calculate_maxtimes.sh`: *Bash* file used to calculate the maximum time that algorithms may be run.
- `run_ex2.sh`: Runs all the experiments of this implementation exercise and prints results to files.
- `analysis_ex2.r`: Reads the result files of this implementation exercise 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 *.c -o SCP
```

This should result in a binary file called `SCP`. The resulting binary can then be called by some parameters. For this implementation exercise, the following are relevant:

- `--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.
- `--ils` or `--aco`: The implemented algorithm to use. On of these is needed to execute an algorithm implemented for this exercise.
- `--mt`: Maximum time to run `--ils` or `--aco`.
- `--mc`: Cost at which to stop iterating.
- `--co`: Cut-off time when using `--mc`.
- `--trace`: File to which better costs with their time of achieving needs to be written.

To execute the experiments, one first needs to calculate the maximum time that an algorithm may run. To calculate this and write the results to a file, the following command is needed:

```
./calculate_maxtimes.sh
        PATH/TO/SCP
        PATH/TO/INSTANCES/DIRECTORY
        /PATH/TO/maxtimes.txt
```

Once this is done, the bash file to execute the actual experiments can be run using the following command:

```
./run_ex2.sh
        PATH/TO/SCP
        /PATH/TO/maxtimes.txt
        PATH/TO/INSTANCES/DIRECTORY
        /PATH/TO/best-known.txt
        PATH/TO/RESULTS/DIRECTORY
```

These are all relative paths. For the first experiment, each algorithm is run on each instance for a certain time, which is the one specified in `maxtimes.txt`. The resulting costs are then written to a text file specifically for that algorithm in the results directory.

For the second experiment, both algorithms were run 25 times on instances *A.1*, *B.1*, *C.1* and *D.1*. Using the necessary parameters, trace files were written to the results directory.

When software to execute *R* code is installed, the file `analysis_ex2.R` can be executed to analyze the results obtained by running the `run_ex2.sh` file. In the command line, this can be called like this:

```
Rscript analysis_ex2.R
        PATH/TO/best-known.txt
        /PATH/TO/maxtimes.txt
        PATH/TO/RESULTS/DIRECTORY
```

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

4 Results

During execution of the algorithms, several values for metrics were obtained. All the obtained costs and the solution qualities of the results of both algorithms on all instances can be found in Appendix A. Other results can be found in the attached ‘results‘ folder.

We first look at the correlation plot for the percentage deviations from the best cost of the execution of both algorithms on each instance. As mentioned in Section 3, the costs were obtained after running for exactly the time for the specific instance mentioned in the `maxtimes.txt` file. This can be seen in Figure 1.

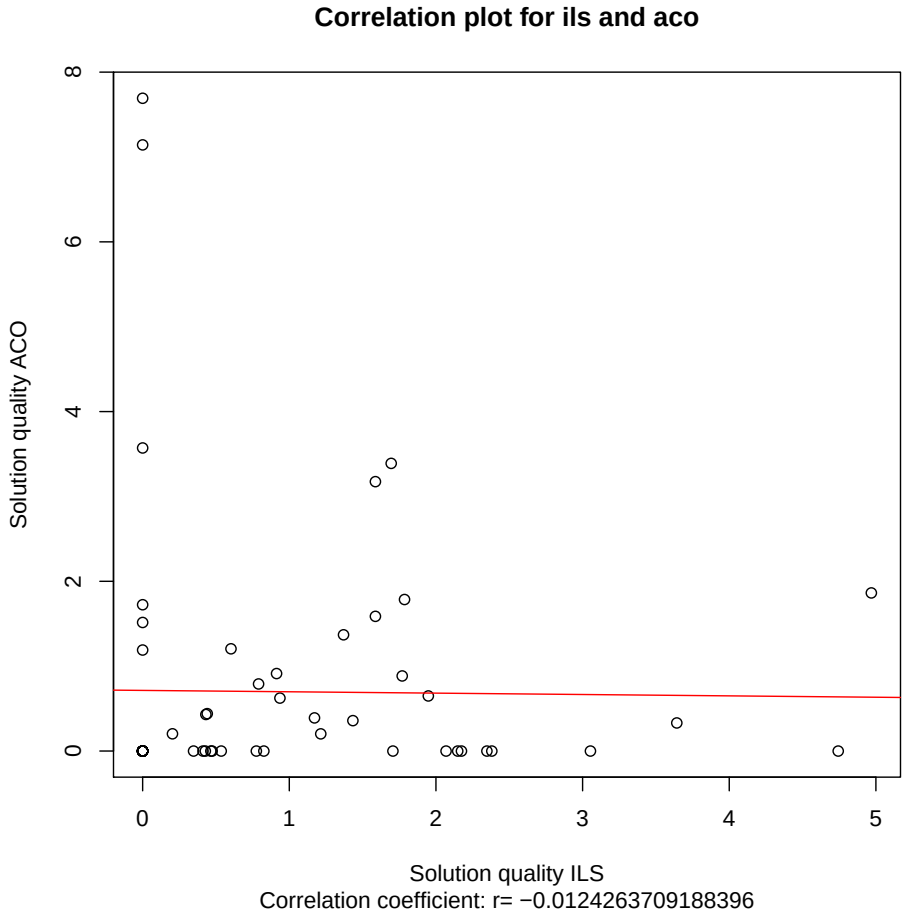

Figure 1: Correlation plot for the solution qualities of the ILS and ACO results.

It can be seen visually that there is no correlation between the results of both algorithms. This can be verified by the correlation coefficient shown under the plot, which is close to 0. Some percentage deviations are high for one algorithm and low for the other. These visually big differences are however sometimes the result of small differences on instances with already a low best cost, which can result in a quite large percentage deviation (for example: a cost of 15 is a deviation of 7,14% from the optimal cost 14). This effect is lower for small differences on instances with a higher best cost.

We now use a *Wilcoxon* test to see if there is a statistically significant difference between the relative percentage deviations of the results of both algorithms. Using this, we get a P value of 0.120. Using a significance level of 0.05, we can say that there is no significant difference between the relative percentage deviations of the results of both algorithms. The average deviations for ILS and ACO were respectively 0.861 and 0.701.

To see how quickly the algorithms reach a 2% deviation from the optimum, we use qualified run-time distributions. For a certain algorithm and instance, we execute the it 25 on that instance, stopping each time when the cost threshold is reached ($1.02 * optimum$) or when it has run for 10 times the already calculated maximum times. Each run used as a seed $10000 + run$, in which run is in the range 1–25.

We first look at the results of ILS on instances *A.1*, *B.1*, *C.1* and *D.1*, which can be seen in Figure 2.

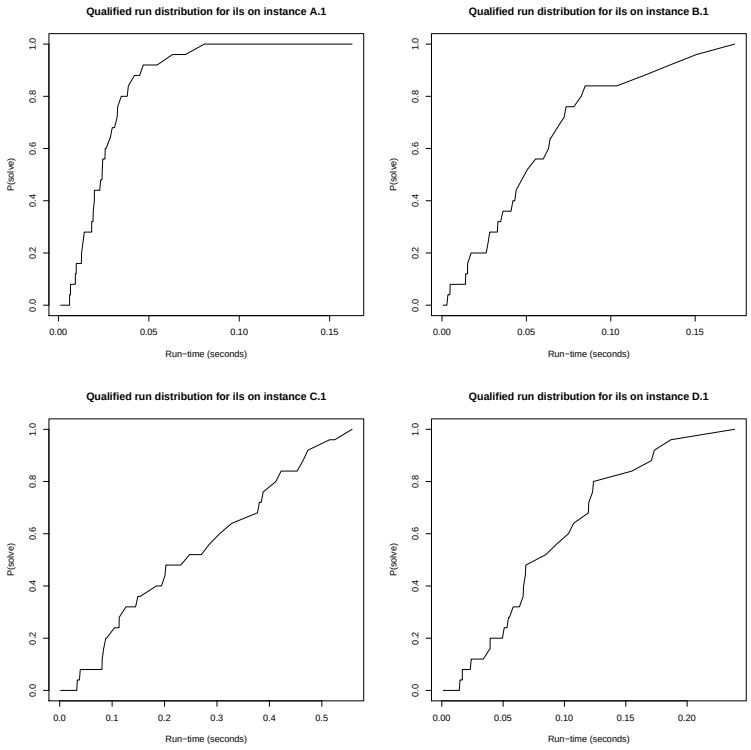

Figure 2: Qualified run-time distributions for algorithm ILS on instances *A.1*, *B.1*, *C.1* and *D.1*.

We can see that the time until every run reaches the cost threshold depends on the instance on which the algorithm is run. Algorithm *C.1* seems to need the most amount of time. It can also be noticed that, in general, the chance of solving increases the most at the start and is the lowest at the end. This means that some runs use a "bad" seed that results in a longer than average time until reaching the cost threshold.

The results of ACO on instances the same four instances can be seen in Figure 3.

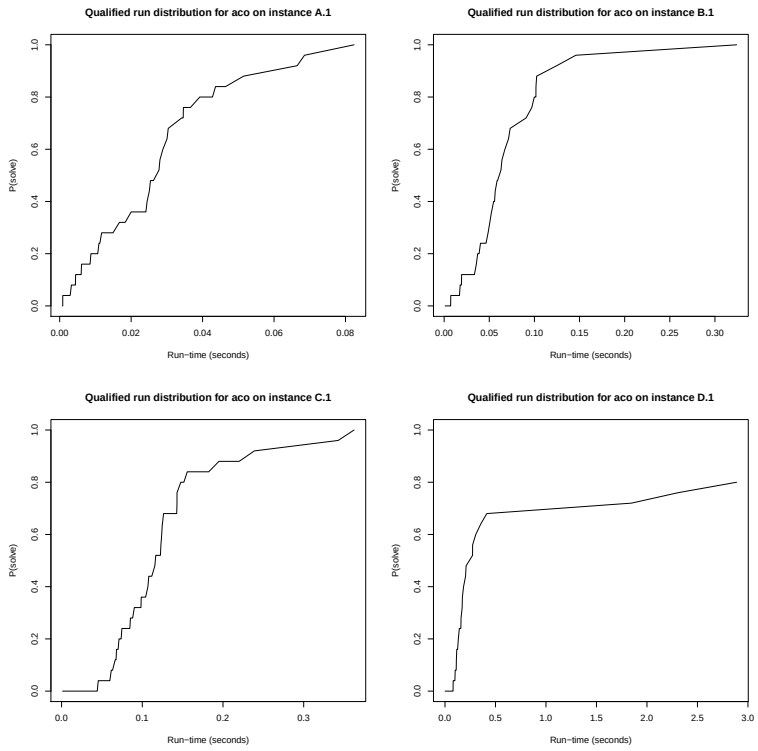

Figure 3: Qualified run-time distributions for algorithm ACO on instances *A.1*, *B.1*, *C.1* and *D.1*.

We see that these results are about the same as those of ILS in Figure 2, apart from the result on instance *D.1*. Some runs seem not to reach the threshold in time with that instance. As can be seen, only about 20 runs do reach the cost threshold. This means that the seed has a lot of influence on the time until reaching the threshold (or maybe reaching the threshold at all)

when executing ACO on instance *D.1*. This seed is used when choosing an element to be covered and choosing a set with a certain probability, both in the construction phase. Thus, as a result, the pheromone trails and heuristic information are also influenced by this. These may then possibly enforce the effect of the seed.

This problem wasn't noticed on other instances, even with different seeds as those those previously described.

5 Conclusion

First, the algorithms Iterated Local Search and Ant Colony Optimization were described as I implemented them. In the results, we saw that the results of both algorithms are uncorrelated, which could be due to the way the solution quality is calculated. When we used the *Wilcoxon* test, we noticed that the differences in the solution qualities of both algorithms are insignificant. When looking at the qualified run-time distributions, we saw that for 3 out of the four instances the behavior of both algorithms was almost the same. Only the result of ACO on instance *D.1* deviated. Some runs didn't reach the cost threshold on time. This could be caused by the seed that is used for random picking.

References

Jacobs, L. W. and Brusco, M. J. (1995). Note: A local-search heuristic for large set-covering problems. *Naval Research Logistics (NRL)*, 42(7):1129–1140.

Ren, Z.-G., Feng, Z.-R., Ke, L.-J., and Zhang, Z.-J. (2010). New ideas for applying ant colony optimization to the set covering problem. *Computers & Industrial Engineering*, 58(4):774–784.

Appendices

A Costs and solution qualities

	ILS	ACO

Instance	Optimal	Max. runtime (sec.)	Cost	Solution quality	Cost	Solution quality
4.2	512	2.05	523	2.148	512	0.000
4.3	516	2.02	520	0.775	516	0.000
4.4	494	2.00	500	1.215	495	0.202
4.5	512	1.95	518	1.172	514	0.391
4.6	560	1.98	563	0.536	560	0.000
4.7	430	1.87	432	0.465	430	0.000
4.8	492	1.96	493	0.203	493	0.203
4.9	641	2.04	647	0.936	645	0.624
5.1	253	2.51	265	4.743	253	0.000
5.2	302	2.46	313	3.642	303	0.331
5.3	226	2.45	230	1.770	228	0.885
5.4	242	2.46	244	0.826	242	0.000
5.5	211	2.40	212	0.474	211	0.000
5.6	213	2.45	218	2.347	213	0.000
5.7	293	2.52	298	1.706	293	0.000
5.8	288	2.51	289	0.347	288	0.000
5.9	279	2.40	283	1.434	280	0.358
6.1	138	1.97	141	2.174	138	0.000
6.2	146	1.94	148	1.370	148	1.370
6.3	145	2.03	148	2.069	145	0.000
6.4	131	1.96	135	3.053	131	0.000
6.5	161	1.96	169	4.969	164	1.863
A.1	253	3.69	255	0.791	255	0.791
A.2	252	3.70	258	2.381	252	0.000
A.3	232	3.66	233	0.431	233	0.431
A.4	234	3.74	235	0.427	234	0.000
A.5	236	3.68	236	0.000	236	0.000
B.1	69	3.70	69	0.000	69	0.000
B.2	76	4.08	76	0.000	76	0.000
B.3	80	3.76	80	0.000	80	0.000
B.4	79	4.01	79	0.000	79	0.000
B.5	72	3.86	72	0.000	72	0.000
C.1	227	5.65	228	0.441	228	0.441
C.2	219	5.37	221	0.913	221	0.913
C.3	243	5.30	244	0.412	243	0.000
C.4	219	5.29	219	0.000	219	0.000
C.5	215	5.24	215	0.000	215	0.000
D.1	60	5.60	60	0.000	60	0.000

D.2	66	5.69	66	0.000	67	1.515
D.3	72	5.84	72	0.000	72	0.000
D.4	62	5.82	62	0.000	62	0.000
D.5	61	5.85	61	0.000	61	0.000
NRE.1	29	8.50	29	0.000	29	0.000
NRE.2	30	10.44	30	0.000	30	0.000
NRE.3	27	10.39	27	0.000	27	0.000
NRE.4	28	10.54	28	0.000	29	3.571
NRE.5	28	10.91	28	0.000	28	0.000
NRF.1	14	12.91	14	0.000	14	0.000
NRF.2	15	12.74	15	0.000	15	0.000
NRF.3	14	13.03	14	0.000	15	7.143
NRF.4	14	12.91	14	0.000	14	0.000
NRF.5	13	14.93	13	0.000	14	7.692
NRG.1	176	25.13	176	0.000	176	0.000
NRG.2	154	24.33	157	1.948	155	0.649
NRG.3	166	24.75	167	0.602	168	1.205
NRG.4	168	22.91	171	1.786	171	1.786
NRG.5	168	24.74	168	0.000	170	1.190
NRH.1	63	33.20	64	1.587	64	1.587
NRH.2	63	37.35	64	1.587	65	3.175
NRH.3	59	31.38	60	1.695	61	3.390
NRH.4	58	30.42	58	0.000	59	1.724
NRH.5	55	30.37	55	0.000	55	0.000