Heuristics - SCP Paper

Felipe Coutinho

October 30, 2024

1 Introduction

In this work, we will explore methods to solve the Set Covering Problem (SCP). This is a fundamental optimization problem with application to areas such as logistics, and telecommunications. For example, in the former, the SCP helps minimize costs by efficiently placing facilities to cover demand across geographic areas [1]. In the latter, it can be used for efficient coverage in wireless systems, where selecting the minimum number of towers ensures full coverage with minimal overlap and cost [5].

1.1 Problem statement

Consider a set $\mathcal{A}$ with cardinality $|\mathcal{A}| = n$, and a set $\mathcal{F} = \{\mathcal{F}_j, j = 1, \ldots, m\}$ such that each $\mathcal{F}_j \subset \mathcal{A}$ has a cost $c_j \in \mathbb{N}^+$. A solution s is a collection of indices such that $\bigcup_{j \in s} F_j = \mathcal{A}$. Our goal is to find s^* such that the total cost is minimized:

$$s^* = \arg\min_s \sum_{j \in s} c_j \tag{1}$$

1.2 Representation

Construct a binary matrix A with $a_{ij} \in \{0, 1\}$, and $i = 1, \ldots, n$ representing number of elements, and $j = 1, \ldots, m$ representing number of subsets. Each column (subset) has a cost $c_j > 0$.

There are two straight-forward ways to represent a solution s. The first option is to define s as a list of integer indices, corresponding to the columns of A (subsets) to be included in the covering:

$$s_j \in \{1, 2, \ldots, m\} \tag{2}$$

A downside of this representation, from a programming point of view, is that the length $|s|$ is not known *a priori*. The second option is to define s as a binary vector of length $|s| = m$; each s_j indicates whether the j-th column of A is to be included:

$$s_j \in \{0, 1\},\ 0 \leq j \leq m \tag{3}$$

We will use both representations throughout this work.

A strength of either approach is that checking if a given solution is feasible can be done efficiently:

$$\sum_{j \in s} a_{ij} = 1, \forall i \tag{4}$$

2 Constructive heuristics

We propose 3 different constructive heuristics; we start with $s = \varnothing$, adding subsets progressively until full coverage. The overall algorithm pseudocode is given by 2

Algorithm 1 Constructive heuristic

```
$s \leftarrow \{\}$                                      ▷ Initialize solution
while $s$ is not feasible do
    Select $j$ with a dispatching rule                   ▷ See section 2.1
    Include $j$ in $s$
end while
```

2.1 Dispatching rules

The decision of which column to add at each step is made through a dispatching rule.

1. **Greedy heuristic:** At step t, pick a column j^* such that $j^* = \arg\max_j \frac{N^{(t)}}{c_j}$, where $N^{(t)}$ is the number of *new* elements that adding j to s would cover.

2. **Randomized greedy heuristic:** Define threshold γ and probability p. For this work, we selected $\gamma = 0.8$ and $p = 0.1$. At step t, sample $u^{(t)} \sim \text{Uniform}(0, 1)$.

   - If $u^{(t)} > p$, pick a column j^* such that $j^* = \arg\max_j \frac{N^{(t)}}{c_j}$.
   - Otherwise, build set $J = \left\{ j : \frac{N^{(t)}}{c_j} \geq \gamma \times \frac{N^{(t)}}{c_j^*} \right\}$. Then, pick a random sample from J.

3. **Greedy heuristic with priority** [4]**:** It seems reasonable to prioritize less frequent elements earlier. At step t, for each uncovered element i, compute $\eta_i^{(t)}$ - the number of available columns covering it. Let $\sigma[\eta_i^{(t)}]$ be the standard deviation of $\eta_i^{(t)}$, and $\sigma_0 = 5$ a user-defined threshold.

   - If $\sigma[\eta_i^{(t)}] < \sigma_0$, there are no critical elements. Pick a column j^* such that $j^* = \arg\max_j \frac{N^{(t)}}{c_j}$.
   - Otherwise, for each available column j, assign a priority score $\beta_j^{(t)} = -\log \sum_{i \in A_j} \eta_i^{(t)}$. Pick a column j^* such that $j^* = \arg\max_j \frac{N^{(t)} \beta_j^{(t)}}{c_j}$.

2.2 Redundancy elimination

A redundant subset is a member of s which can be safely removed without hurting the solution's feasibility (i.e. the elements covered by it are also covered by other members of s).

Our implementation is given by algorithm 2. Exhaustively searching all subsets of s for optimal redundant removal is too expensive (power set size scales with $2^{|s|}$). Instead, we try to maximize the total cost reduction by removing the redundancy with the largest product $\omega_j C_j$. The value ω_j represents the number of covered attributes overlapping with other members of s.

Algorithm 2 Redundancy elimination

```
Require: $s$                                                        ▷ Solution
Ensure: $\sum_{j \in s} a_{ij} = 1, \forall i$                      ▷ Must be feasible
  $s' \leftarrow$ redundant members of $s$
  while $|s'| \neq 0$ do
      $R \leftarrow \{\}$                                             ▷ Rank each candidate for removal
      for $j \in s'$ do
          $\omega_j \leftarrow$ sum of overlaps between $j$ and $s$
          $C_j \leftarrow$ cost of $j$
          $R \leftarrow \omega_j \times C_j$
      end for
      Remove $\arg\max(R)$ from $s$
      $s' \leftarrow$ redundant members of $s$                      ▷ Repeat until no more redundancies
  end while
```

2.3 Results

The results are summarized below (mean ± standard deviation of relative error from optimal known solution). For each algorithm, we report performance after and before redundancy elimination.

1. **Greedy:** $(13.6 \pm 7)\% \rightarrow (5.5 \pm 6)\%$
2. **Randomized Greedy:** $(14.9 \pm 7)\% \rightarrow (6.1 \pm 6)\%$
3. **Priority Greedy:** $(14.2 \pm 7)\% \rightarrow (6.3 \pm 7)\%$

We can clearly see the benefit of using the redundancy elimination algorithm (approximately 50% increase in average performance).

Regarding the runtime, the constructive heuristic takes around 30 sec. The redundancy elimination needs 5 sec.

3 Improvement heuristics

To find better solutions, we treat the constructive heuristic output as an initial solution to be improved by some *local search* (LS) method.

This means we need to build some neighborhood N around the solution by applying some *move* to it. If the move is too timid, N might be too small to enable the LS to escape local minima. Moreover, some moves might create neighborhoods with unfeasible solutions.

3.1 Search strategy

The first approach was to create $N(1)$ by applying a swap$_{1,1}$ move. This means removing one member of s, and inserting another. However, the solution diversity in $N(1)$ was not great, and the LS could not find improvements to the initial solution beyond redundancy elimination.

Instead, we adopted the swap$_{1,2}$ move (remove one, insert two). Obviously, most neighbors will *not* improve s, since we are likely to add redundancies. Therefore, after each move, we apply the RE routine.

For any given solution, there are $|s| \times \frac{m(m-1)}{2}$ possible moves. This is a rather large neighborhood; for example, $|s| = 50$ and $m = 1000$ correspond to $\approx 25 \times 10^6$ neighbors. This is problematic, specially because we plan to apply RE many times. Therefore, instead of generating the complete neighborhood, we restrict each to $N_s \sim 100$ neighbors .

Additionally, we implement two strategies for stopping the LS at each iteration:

1. **Best-improvement (BI):** look at all N_s neighbors, and pick the best improvement. If there are no improving neighbors, keep the current solution.

2. **First-improvement (FI):** as soon as an improvement is found, move on to the next neighborhood.

Of course, the best-improvement strategy is computationally more expensive, but should lead to better solutions.

Throughout the experiments, we use $N_s = 100$, and perform a maximum of $N_t = 1000$ LS steps. The LS algorithm pseudocode is given by 3.1 and 3.1.

Algorithm 3 First-improvement Local Search

```
Require: $s$                                              ▷ Initial solution
Require: $N_t$                                            ▷ Max. search steps
Require: $N_s$                                            ▷ Neighborhood size per step
  for $0 \leq i \leq N_t$ do
    $\eta^* \leftarrow s$                                 ▷ Best neighbor
    for $0 \leq j \leq N_s$ do
      $\eta \leftarrow \mathrm{swap}_{1,2}$               ▷ Apply swap move to solution
      if $\eta$ is not feasible then
        Skip neighbor
      end if
      Apply redundancy elimination to $\eta$
      if $\eta$ costs $< \eta^*$ then
        $s \leftarrow \eta$
        Break                                             ▷ Restart search with improved solution
      end if
    end for
  end for
```

Algorithm 4 Best-improvement Local Search

```
Require: $s$                                              ▷ Initial solution
Require: $N_t$                                            ▷ Max. search steps
Require: $N_s$                                            ▷ Neighborhood size per step
  for $0 \leq i \leq N_t$ do
    $\eta^* \leftarrow s$                                 ▷ Best neighbor
    for $0 \leq j \leq N_s$ do
      $\eta \leftarrow \mathrm{swap}_{1,2}$               ▷ Apply swap move to solution
      if $\eta$ is not feasible then
        Skip neighbor
      end if
      Apply redundancy elimination to $\eta$
      if $\eta$ costs $< \eta^*$ then
        $\eta^* \leftarrow \eta$                          ▷ Update best solution found so far
      end if
      $s \leftarrow \eta^*$                               ▷ Restart search with improved solution
    end for
  end for
```

3.2 Results

We study how each LS strategy improves initial solutions given by certain constructive heuristics. The results are shown in Table 3.2.

Note that there are no significant differences in runtime between FI and BI strategies. This happens because $N_s = 100$ are small neighborhoods. Hence, it is rare to find even a single improving solution - which causes the FI algorithm to almost always exhaustively search N. Therefore, we might be better off increasing N_s, at the cost of more compute time; if we consider $100s$ per instance as an acceptable worst-case computational time, we might improve our LS by setting $N_s \to 1000$.

This is also reflected in the similar performance of the best solutions found by FI and BI. In general, FI performs worse than BI as expected, but the difference is small.

Another interesting reflection is that throughout the LS, $< 5\%$ of the generated neighbors were feasible. Hence, most neighbors are discarded. This means our swap$_{1,2}$ move damages the solution too much, and we could search more efficiently if we refined it to be more likely to preserve feasibility.

We can see that LS always improve the initial solution *if redundancies are not eliminated beforehand*. Most interestingly, this can lead to better results (e.g. randomized + BI); applying our RE might be too greedy, and cause us to be stuck deeper into local optima.

	Greedy		**Randomized**		**Priority**		**Greedy + RE**	
	First	Best	First	Best	First	Best	First	Best
Runtime (s/inst.)	11.9	12.5	12.3	12.7	12.2	12.3	12.3	12.1
Avg. Err. (%)	3.9	3.5	4.2	**3.3**	3.9	3.9	3.5	3.6
LS Impr. (%)	100	100	100	100	100	100	92.9	90.5

Table 1: Results for local search implementation. We report the runtime per instance, the average error relative to optimal solution, and the percentage of instances where local search helps improving the initial solution.

Finally, we perform pair-wise Wilcoxon tests between each algorithm to look for statistically significant differences between their average results. Refer to table 3.2.

The Wilcoxon test is appropriate because we don't have any hints towards assuming normally distributed solution costs C_s, so we stick to non-parametric approaches. We use the paired observations $(C_{s,i}^{(n)}, C_{s,j}^{(n)})$ for algorithms i, j on the n-th instance.

		Greedy		**Random**		**Priority**		**Greedy + RE**	
		Best	First	Best	First	Best	First	Best	First
Greedy	Best	-	0.011	0.596	0.012	0.276	0.188	0.862	0.598
	First	0.011	-	0.039	0.712	0.891	0.670	0.015	0.015
Random	Best	0.596	0.039	-	0.003	0.219	0.234	0.865	0.403
	First	0.012	0.712	0.003	-	0.499	0.237	0.015	0.050
Priority	Best	0.276	0.891	0.219	0.499	-	0.336	0.310	0.196
	First	0.188	0.670	0.234	0.237	0.336	-	0.400	0.227
Greedy + RE	Best	0.862	0.015	0.865	0.015	0.310	0.400	-	0.748
	First	0.598	0.015	0.403	0.050	0.196	0.227	0.748	-

Table 2: Wilcoxon test p-values. Statistically significant results ($\alpha = 5\%$) are highlighted in green.

It is clear that the Greedy Heuristic with Priority does not improve significantly on any other algorithm. Hence, we are better sticking to simple Greedy or Randomized Greedy approaches.

Focusing on the comparison between Greedy and Greedy with RE heuristics, we can see that it is better to apply RE beforehand when using the FI strategy.

4 Escaping local optima

We now turn our attention to neighborhood-based metaheuristics to attempt to escape local optima. The selected methodologies are the Simulated Annealing and the Variable Neighborhood Search.

4.1 Simulated Annealing

The Simulated Annealing (SA) is a very popular metaheuristic. By adopting a probabilistic acceptance criteria, it promotes diversification at early stages (allowing a thorough exploration of the solution space), while shifting towards intensification later on.

This is achieved by selecting some move (which will be applied to an initial solution to create new candidates), and recentering the search with probability $e^{-\frac{\Delta\nu}{T}}$, where ν is an evaluation function and T the temperature. With proper choice of initial temperature T_i, there is a chance for accepting candidates which deteriorate the solution, in an effort to escape local minima. By reducing T according to some cooling schedule, only improving candidates are accepted.

4.1.1 Initial considerations

Before deciding on a particular realization of SA, we reflect on the results obtained in 3. Some important remarks are due:

- The representation used so far corresponds to (2). As the algorithms were implemented with `Python` using the `Numpy` package, it turns out that using (3) instead leads to significant runtime improvement.
- The swap$_{1,2}$ move used in the local search (see 3) implies a large neighborhood ($\sim 10 - 1000 \times 10^6$), which is costly to completely explore. Moreover, we noticed that such move tended to created unfeasible solutions very often ($\gtrsim 95\%$) if the swaps are randomly allocated.

Because of this, we change the representation to (3), and instead of simply rejecting unfeasible solutions, we introduce a *repair mechanism*, together with a *penalization term* in our evaluation function. The pseudocode is given in 4.1.1. We hope that by allowing partial solutions, we will improve the diversification. Note that the penalization term is necessary to distinguish between complete solutions, and similar partial solutions which were repaired. Hence, we guide the underlying algorithm towards the feasible region of the solution space. Throughout this work, we set $\lambda = 1.25$ [6].

Algorithm 5 Repair mechanism for partial solutions.

```
Require: $s$                                                      ▷ Partial solution
Require: $\omega$                                                 ▷ Cost array
Require: $\lambda > 1$                                            ▷ Penalty multiplier
  $\omega_0 \leftarrow \sum_{j \in s} s_j$                         ▷ Initial cost
  $\gamma \leftarrow 0$                                           ▷ Accrued penalty
  while $s$ is partial solution do
      for each $j \in 1, \dots, m$ ordered by increasing $\omega_j$ do
          if $j$ covers any uncovered element in $s$ then
              $s \leftarrow s \cup j$
              $\gamma \leftarrow \gamma + \lambda\omega_j$
          end if
      end for
  end while
  $\omega_T \leftarrow \omega_0 + \gamma$                    ▷ Total cost return $s, \omega_T$
```

4.2 SA Implementation

We adopted the commonly used geometric cooling schedule, parameterized with a cooling rate α: $T_n = \alpha T_{n-1}$.

The candidates are generated by applying a random swap$_{1,2}$ move (remove one, insert *up to* 2); partial solutions are repaired and penalized through 4.1.1.

Some approaches were considered for selecting T_i. For example, one can pick:

$$T_i = \arg\min_T |\mathbb{E}[\nu_1](T) - \nu_0| \tag{5}$$

...where ν_1 is the evaluation function result after first step. The intuition behind this was that preserving the expected fitness would help to avoid the solution degrading too much at the start [2]. However, we found two major problems with this approach:

1. For some classes of instances, the T_i still lead to divergence early on
2. To compute the expectation, we must average over all possible swap$_{1,2}$ moves, which scales poorly as discussed previously. Therefore, this initialization procedure was too slow.

Other strategies involve computing σ - the standard deviation of $\Delta\nu$ - for the first annealing step. Then, set $T_i = k\sigma$ with $k \sim -3/\log p$, where p is the initial probability of acceptance [7]. We tried this approach for different p, but ultimately found that the initial solution tended to diverge significantly unless $p \ll 0.1$ - which suggests a very timid diversification phase.

Instead, we perform a grid search for T_i with a representative instance from each instance class. This is named *calibration phase*. We select 5 values of T_i between 1 and 100, and run the SA. Then, we recenter in the best performing T_i choice, and perform a finer search around $[T_i - \delta T, T_i + \delta T]$. We use $\delta T = 5$, and search at 5 additional points. This approach was faster, and lead to better behaved SA for all instance classes.

The choice of T_f was fixed at $0.01 \times T_i$, and α was selected accordingly to the desired total number of iterations N:

$$T_f = \alpha^N T_i \Leftrightarrow \alpha = \left(\frac{T_f}{T_i}\right)^{\frac{1}{N}} \tag{6}$$

Finally, we must select how many candidates N_T will be generated. We compared exhausting the N_t trials at each step, against cooling as soon as some candidate as accepted (*non-equilibrium* strategy [7]); we found that both led to similar annealing behaviors, so we opted for the latter as it was computationally cheaper. We also implemented the following adaptive strategy to update N_T, as described in reference [7]:

$$N_T = N_{T,0} + \lfloor N_{T,0} \times F \rfloor \tag{7}$$

... where $F = 1 - \exp -(\nu_{\max} - \nu_{\min})/\nu_{\max}$.

Algorithm 6 Simulated Annealing

```
Require: $s_0$                          ▷ Initial solution
Require: $T_i, T_f$                     ▷ Initial and final temperatures
Require: $N$                            ▷ Number of iterations
Require: $N_{T,0}$                      ▷ Number of iterations for convergence at temperature $T$
  $s \leftarrow s_0$
  while $T < T_f$ do
      $i = 0$
      for $i < N_T$ do
          $\eta \leftarrow \text{swap}_{1,2}(s)$
          if $\eta$ is partial solution then
              repair $\eta$
          end if
          eliminate redundancies $\eta$
          $u \sim \text{Uniform}(0, 1)$
          if $u < e^{-\frac{\Delta\nu}{T}}$ then
              $s \leftarrow \eta$
              $T \leftarrow \alpha \times T$
              break
          end if
          $i \leftarrow i + 1$
          update $N_T$
      end for
  end while
```

4.3 Results

The calibration phase results are presented together with the performance per instance class in table 4.3. We set $N = 100$ and $N_{T,0} = 1000$, with a corresponding $\alpha \in [0.95, 0.99]$. We call a *hit* whenever the solution matches the best known solution. Finally, an example of the

SA evolution is shown in figure 1. The total run time for the calibration and SA procedures, over all instances, sum up to 150 minutes (around 3.5 minutes per instance).

Instance Class	4	5	6	a	b	c	d
T_i	30	3.5	1.0	6.0	1.0	3.5	1.0
Error (%)	3.9	4.6	8.2	4.6	3.2	4.2	8.7
Hits	0	0	0	0	0	0	0

Table 3: Simulated Annealing results per instance class.

Unfortunately, comparing to the previous results from section 3, we can see that there was no significant improvement from the SA approach. We can draw some conclusions:

1. The swap$_{1,2}$ causes the cost to oscillate with the same amplitude throughout the whole process. This means that the local search stage (at low temperatures) is very inefficient, since it spends most of the compute time considering very bad solutions.

2. Although the swap$_{1,2}$ move is flexible enough to allow for variable $|s|$, it seems that it failed to introduce enough diversity. Perhaps, when combined with the repairing procedure, it creates cycles where we visit the same restricted set of local optima repeatedly.

3. Perhaps adaptive cooling schedules are more appropriate for this problem, as figure 1 suggests most of the time is spent during intensification, without very noticeable payback.

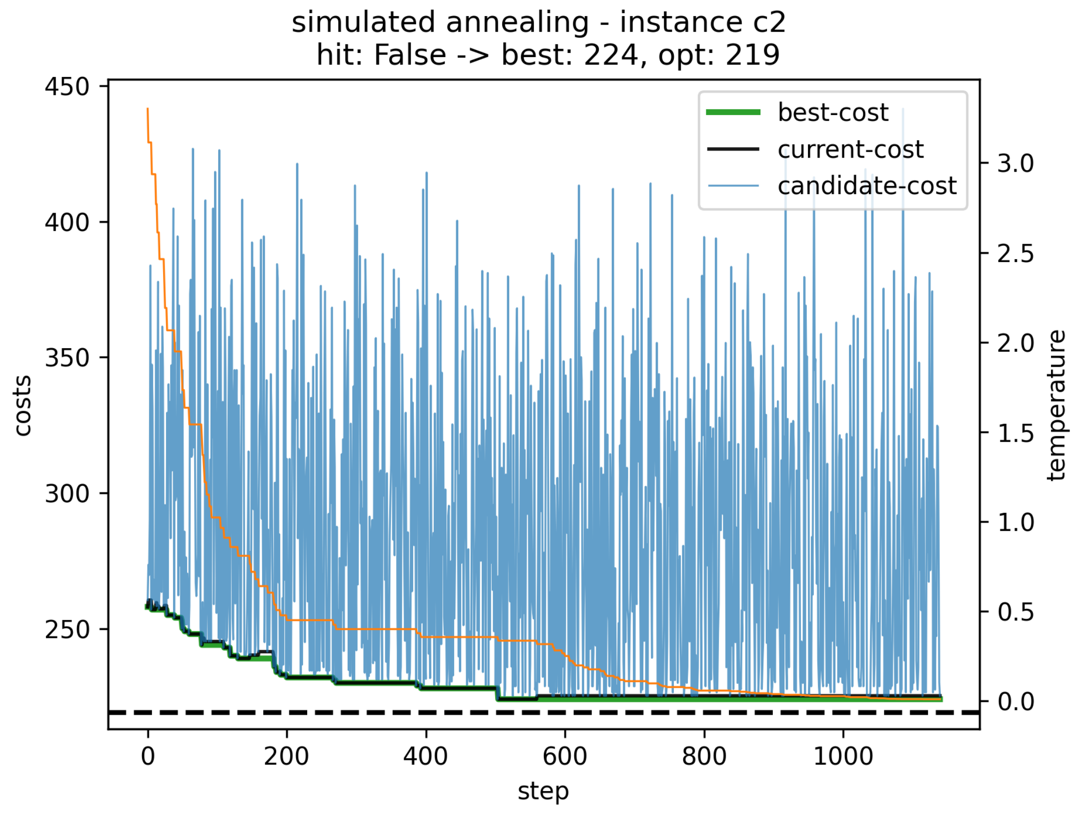

Figure 1: SA algorithm example. Notice how at each temperature, as soon as a new candidate is accepted, we apply the cooling schedule. The proposed candidates cause the cost to oscillate with large, constant amplitude throughout the whole procedure.

4.4 Variable Neighborhood Search

The Variable Neighborhood Search (VNS) has a very generic structure, which lend it useful for many different types of applications.

In essence, it pairs some shaking strategy with a local search procedure, to look for local minima in an organized way. The shaking strategy consists in applying some move to a local minimum such that a new starting point for LS is obtained; as it is unknown *a priori* how strong this perturbation must be to escape a valley, the shaking employs some nested neighborhood structure $N_k,\ k = 1, \ldots, k_{\max}$. After each unsuccessful attempt of finding a better minimum through LS, the next shaking happens in a larger space.

4.5 Implementation

Following the discussion of section 4.1, we keep the binary solution representation of (3). We decide to also use the swap$_{1,2}$, due to its flexibility in allowing $|s|$ to change.

Since this move required the repair mechanism to consistently propose feasible solutions, we keep it for the shaking step.

In order to build the nested neighborhood structure, we sequentially apply the move k-times [3]. To decide on the value for $k_{\max}$, a preliminary study was carried out with some representative instances from each class. We selected $k_{\max} = 10$, which we considered to be excessively large; then, executing the algorithm for fixed $N = 1000$ iterations at each k, we observed how often improvements were found at each stage. Most improvements were found at most with $k_{\max} = 3$, so we decided to keep $k_{\max} = 5$ to allow some leeway when running the procedure with larger N.

For the local search step, we choose to *not reutilize the procedures from section 3*. This is because that method uses the same $\text{swap}_{1,2}$ move, so we would be very likely for $k = 1$ to undo the shaking and return to the same local optima. To avoid this cycle, we instead use a new move: flip_1. It consists in randomly selecting a $j \in \{1, 2, \dots m\}$ and flipping the corresponding value in s. In other words, if the selected j is present in s, we remove it; if it is not, we insert it. This LS is applied with a best-improvement strategy, with a fixed total of N trials. The pseudocode is presented in algorithm 7

Algorithm 7 Variable Neighborhood Search

```
Require: $s_0$                                         ▷ Initial solution
Require: $k_{\max}$                                    ▷ Max. neighborhood size
Require: $N$                                           ▷ Local search depth
Require: $M$                                           ▷ VNS iterations
  $s \leftarrow s_0$
  for $M$ times do
      while $k \leq k_{\max}$ do
          for $k$ times do                             ▷ Shaking
              $s' \leftarrow \text{swap}_{1,2}$
          end for
          repair $s'$
          eliminate redundancies of $s'$
          for $N$ times do                             ▷ Local search
              $s'' \leftarrow \text{flip}_1(s')$
              store best $s''$ found so far
          end for
          if best $s''$ found improves $s$ then        ▷ Recenter search in new optimum
              $s \leftarrow s''$
              $k \leftarrow 1$
          else                                         ▷ Shake harder to escape current valley
              $k \leftarrow k + 1$
          end if
      end while
  end for
```

4.6 Results

To pick N, we reason that the local search space has cardinality of m (as there are m possible flips for a given solution). Therefore, we simply match $N = m$ for each instance.

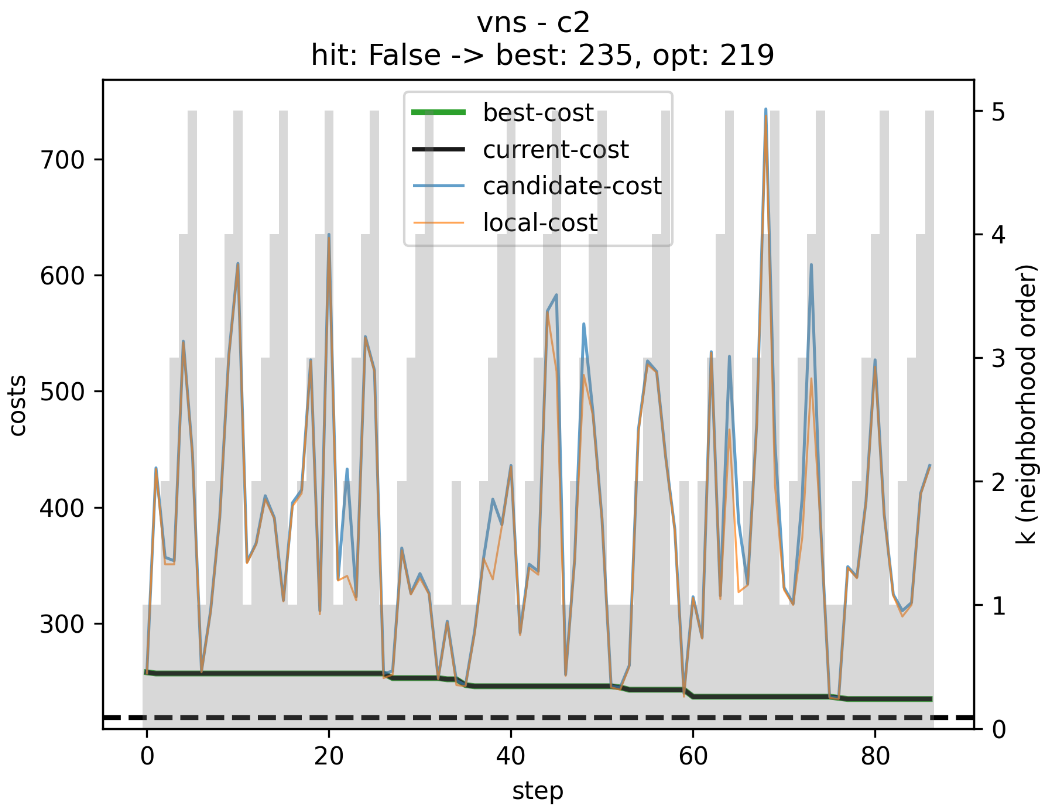

Figure 2: VNS algorithm example. The light gray bars indicate the size k of the neighborhood used for the shaking.

On the other hand, we repeat the VNS loop M times, in order to explore a larger portion of the neighborhoods induced by $\text{swap}_{1,2}$. With some preliminary runs, we choose, $M = 25$ as it corresponds to an average of 4 minutes per instance. For comparison, $M = 100$ led to 15 minutes per instance (which is too slow for our purposes), without substantial improvement in performance.

In table 4.6, we can see the results per instance. Note that the total run time was 700 minutes, which represents around 4.3 minutes per instance. Moreover, figure 2 shows an example of the VNS evolution.

Instance Class	4	5	6	a	b	c	d
Error (%)	5.8	6.2	7.6	9.2	4.3	6.2	8.4
Hits	0	0	0	0	0	0	0

Table 4: VNS results per instance class, for $M = 25$.

Once again, we could not find optimal solutions. The relative errors are similar to SA; a $t-$test with 0.95 confidence between per-instance results, shows that these algorithms are significantly different (95% confidence level). Here are some reasons for the shortcomings of our VNS method:

- When exploring $k > 1$, it is very likely that the solution will be unfeasible, and the repair mechanism will make it have a high cost. This can be seen in figure 2, as the candidate costs are usually much larger than current.
- Moreover, sometimes the local search is unable to find a complete neighbor within flip_1 range; in these cases, it also resorts to repairing, which causes them to return very bad "local optima". This can be seen in 2 as the orange and blue lines are usually very close together (specially for large k).
- Perhaps running it for 25 iterations is too restricting, and the algorithm starts performing much better if we increase the compute time considerably. Notice that this is not expected of SA, as once the solution converges and $T \to 0$, there isn't any mechanism to actively explore other local optima.

5 Conclusions

In this work, we have proposed and evaluated constructive heuristics for the Set Covering Problem. We have also implemented local search procedures, to improve upon these greedy constructive methods. Finally, we attempted to apply well-established metaheuristics to deal with the problem of local minima. Our Simulated Annealing and Variable Neighborhood Search approaches were lacking in regard to overall performance, as they failed to find any of the best-known solutions over the instance set. Further work should be done to find better approaches, such as using different moves and adaptive cooling for SA, or allowing the VNS to run for longer, with more intricate local search methods.

References

[1] Marco Boschetti and Vittorio Maniezzo. "A set covering based matheuristic for a real-world city logistics problem". In: *International Transactions in Operational Research* 22.1 (2015), pp. 169–195.

[2] Eduardo Meca Castro. "Two Neighbourhood-based Approaches for the Set Covering Problem". In: *U. Porto Journal of Engineering* 5.1 (2019), pp. 1–15.

[3] Pierre Hansen and Nenad Mladenovic. "A tutorial on variable neighborhood search". In: *Les Cahiers du GERAD ISSN* 711 (2003), p. 2440.

[4] Guanghui Lan, Gail W. DePuy, and Gary E. Whitehouse. "An effective and simple heuristic for the set covering problem". In: *European Journal of Operational Research* 176.3 (2007), pp. 1387–1403. DOI: 10.1016/j.ejor.2005.09.028.

[5] Mahmoud Owais, Mostafa K Osman, and Ghada Moussa. "Multi-objective transit route network design as set covering problem". In: *IEEE Transactions on Intelligent Transportation Systems* 17.3 (2015), pp. 670–679.

[6] Sandip Sen. “Minimal cost set covering using probabilistic methods”. In: *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*. 1993, pp. 157–164.

[7] EG Talbi. “Metaheuristics: From Design to Implementation”. In: *John Wiley & Sons google schola* 2 (2009), pp. 268–308.