Heuristics - SCP Paper

Felipe Coutinho

October 15, 2024

1 Introduction

In this work, we will explore methods to solve the Set Covering Problem (SCP).

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:

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

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 $c_j > 0$. The solution s is a list with indices for included columns.

With this representation, it is cheap to check if a given solution is feasible: $\sum_{j \in s} a_{ij} = 1, \forall i$.

2 Constructive heuristics

We propose 3 different constructive heuristics; we start with $s = \varnothing$, adding subsets progressively until full coverage. 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:** 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 $j^* = \arg\max_j \frac{N^{(t)}\beta_j^{(t)}}{c_j}$.

3 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 1. Exhaustively searching all subsets of s for optimal redundant removal is too expensive (power set size scales with $2^{|s|}$). Instead, find K the most promising initial sets to prune. A promising set has:

1. Many elements overlapping with the other solution members.
2. High cost.

Then, for each of them, perform a greedy search for additional redundant members of s.

4 Results

The results are summarized below (mean $\pm$ 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.

Algorithm 1 Redundancy elimination

```
Require: $s$
Ensure: $\sum_{j \in s} a_{ij} = 1, \forall i$                ▷ Feasibility check
  $K \leftarrow 5$                                             ▷ Top-K elimination candidates to explore
  $s' \leftarrow$ Prune($s$)                                   ▷ Don't consider elements which are critical for feasibility
  for $j \in s'$ do
      $o_j \leftarrow$ Overlap($j, s'$)                        ▷ Total elements in $j$ that are also in other sets
      $C_j \leftarrow$ Cost($j$)
      Rank($j$) $\leftarrow o_j C_j$                           ▷ Prefer candidates with high cost and/or many overlaps
  end for
  Sort($s'$) by Rank
  $\xi \leftarrow$ Top-K Candidates($s'$, Rank($j$))           ▷ Select most promising candidates
  for $1 \leq k \leq K$ do                                     ▷ Greedy search
      $\alpha_k \leftarrow \xi_k$                              ▷ Start pruning sequence with $\xi_k$
      while feasible do
          $\zeta \leftarrow \arg\max_j C_j, j \in$ IsRedundant(s')   ▷ Find highest cost redundant set
          $\alpha_k \leftarrow \zeta$                          ▷ Append to pruning sequence
      end while
  end for
  $\alpha \leftarrow \arg\max_k \sum_{j \in \alpha_k} C_j$     ▷ Find pruning sequence with highest value
  for $j \in \alpha$ do
      $s^* \leftarrow s \setminus \{j\}$                       ▷ Prune initial solution
  end forreturn $s^*$
```

5 Improvement heuristics

We attempt to improve the initial solutions provided by our constructive heuristics. For this, we use a **N1 scheme** (i.e. generate neighborhoods by applying a single move) **with swaps**. We try to restrict ourselves to the most promising part of the neighborhood.

For this, first select $\rho \in s$ to be removed. Sort the possible $\{\rho\}$ by the number of elements which are uncovered when we remove it. Then, for each ρ, pick some $\alpha, 1 \leq \alpha \leq m$ to be added. Sort the possible $\{\alpha\}$ by the number of elements which are covered when we include it. Prune the possible $\{\alpha\}$ by considering only those where $\Delta = c_\alpha - c_\rho < 0$ (i.e. we expect a decrease in total cost).

For each pair (α, ρ), we make the swap, check for feasibility, and perform redundancy elimination.

In the **best-improvement** strategy, we parse *every* possible (α, ρ) before updating our incumbent solution.

In the **first-improvement** strategy, we stop looking as soon as we find some improvement.

6 Results

Always starting from the same initial solution, we run each improvement heuristic 3 times. The initial solutions are obtained from:

- **Greedy heuristic**
- **Randomized Greedy heuristic**
- **Priority Greedy heuristic**
- **Greedy heuristic + Redundancy elimination**

The results are shown in Table 6. It seems that our local search **fails to improve beyond redundancy optimization**. In fact, we can see that every instance without RE benefits from it; on the other hand, if we apply RE in advance, only 14.3% of instances benefit.

We can see that every algorithm clocks in about 7 minutes (aggregated over all instances). It was expected to see the first-improvement search to have significantly smaller run time. However, since we don't find improvements in the generated neighborhood, the first-improvement ends up searching most of it. Hence, for our initial solutions, our LS procedures are equivalent.

	Greedy		Randomized		Priority		Greedy + RE	
	First	Best	First	Best	First	Best	First	Best
Runtime (s)	423	422	426	427	432	433	388	391
Avg. Err. (%)	5.5	5.5	5.8	6.0	6.3	6.3	5.5	5.5
LS Impr. (%)	100	100	100	100	100	100	14.3	14.3