# Project Report

Solving Optimization Problems with Search Heuristics

MPRI M2

Students: Noémie CATHERINOT, Zacharoula SOTIRIOU

## 1 Introduction

Our project followed a progressive, experimental approach. We started from the idea that random restarts can help escape local optima, then gradually refined and extended the algorithm to study the effect of three main design dimensions:

1. the *restart rate* and the mean *restart budget*;
2. the *distribution* (variance) of the restart budget;
3. the *bias rate* in the two-population version.

We ran all experiments on the graph-based benchmark problems of the IOHprofiler library, including `MaxCut`, `MaxCoverage`, `MaxInfluence`, and `PackWhileTravel`. Each curve in our plots represents the mean of several independent runs ($n_{\text{reps}} = 3$), each with a total evaluation budget of 10,000 (occasionally reduced for slower problems to limit computation time). For comparison, we also plotted classical baselines such as Random Search, Randomized Local Search (RLS), and the $(1 + 1)$ EA.

## 2 Algorithm

### 2.1 Restarts

Our very first idea was to use a variant of $(1 + 1)$EA with restarts. Indeed, if we have a function with (many) local optima, $(1 + 1)$EA very easily gets stuck in one, relying on the very small chance that multiple good flips are done at once.

Even if we take a $(1 + 1)$EA with a small probability of accepting a worse solution, we saw, in the case of the JUMP function that we still have to wait a really long time to jump that gap between the two optima. Of course, in the case of the JUMP function, the restart method is not really useful either as you have to sample directly the global optima as it is isolated. However, we believe that in the general case, optima are more likely to resemble a "mountain" rather than a "lone tower surrounded by a moat". In which case, restarts have a higher efficiency as they allow us to jump between mountains and perhaps land on the right hill. And in the "needle in the haystack" type of functions (the JUMP function could be assimilated to that case if the gap to bridge is large), even the best of algorithms are practically guessing blindly, so our algorithm relying on sampling the right restart point is not too bad.

In any case, we find the result of an algorithm that tried to explore far away by randomly restarting more comforting than that of an algorithm that stayed in a narrow path and explored only one hill. Intuitively, this behavior feels more trustworthy: the algorithm has explored multiple regions before committing, providing a broader sense of confidence in the result.

The problem with restarts, is, of course, time. If the current region already contains a global optimum, restarts waste valuable evaluations that could have been used for local improvement. Even if we were not on the right peak of the graph, restarts have an unknown probability of being fruitful (depends on the number of "right" peaks (local optima) and their width).

Therefore we have to limit the number of restart we do. Our algorithm has a *restart rate* parameter which allows to control that.

The number of restarts we do is not the only thing we can play with/have to limit. We have to give a little bit of time to a restart to see if it shows promise or if we should go back to exploiting our current best solution. It is a sort of "probationary period" for the restart. We have to allow enough time for it prove itself useful, but not too much as to not slow down the algorithm. We therefore allocate a budget to the restart.

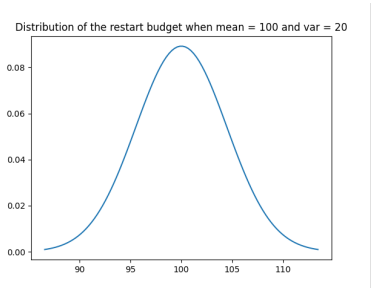


Figure 1: Distribution of the restart budget when its mean = 100 and its variance = 20

## 2.2 Budget of a restart

When the algorithm launches a restart, the first thing we do is determine its budget. Our idea was for the budget to be sampled from a normal law. We therefore have two parameters to set for our budget : the mean of what we think would be a healthy budget, and how much we would like to be able to differ from that mean.

When that allocated budget has run out, if the restart attempt has given us a better candidate than before, we continue on that path. Otherwise, we resume exploiting the best candidate.

However, we do not necessarily toss our efforts aside. If the restart, although not as good as the other candidate, shows promise, we can store it away and give it a little bit more time during our execution to try and see if there is something to be found there.

## 2.3 Population size

Having a population of candidates to draw from rather than just exploring the current best means having multiple pathways being exploited at once.

Our idea is to have at least two axis of research. If a restart, even if it has not managed to beat the current best, has at least achieved a better result than the second search path, we can store it there.

We have to determine empirically what size of population seems to yield better results. Again, the problem with this method is that it slows down the algorithm as we allocate time away from the best-so-far solution.

To remedy that, we implement a bias.

## 2.4 Bias towards the favorite

Say we have two axis of research (which is what we chose). We call them "favorite" and "backup".

Our algorithm has a bias parameter; at each iteration it draws and with large probability decides to update the favorite.

However, once in a while the backup is selected for update. We produce a child from the backup using our mutation operator, and if the child is fitter than its parent, it becomes the new backup candidate. We also check if that child is fitter than the favorite : if it is, it becomes the new favorite and the former favorite becomes backup.

This backup stores the "second best region to explore". If a restart produced a candidate that is not as good as the favorite but is still interesting in the sense that it is at least as good as the backup, we replace the old backup by this new candidate.

## 2.5 Mutation operator

We use a very basic bit-wise mutation operator with mutation rate 1/(number of bits). We did not play try changing this mutation rate to much as there are already theory results backing up this specific rate and it was not the main interest of our algorithm.

## 2.6 Plateaus

The library does this :

Meaning that it only updates the current best when a solution of strictly better fitness is provided. We thought this would create a problem when reaching a plateau as we would not move for a long time until we are able to jump the distance off that plateau. Therefore, if a child is equally as fit as its parent, we keep the child, hoping that wandering the plateau will get us out of it faster.

```
Problem is LeadingOnes
-------------------------------------
Test on [1,0,0,0,0] : result = 1.0
Test on [1,0,1,0,0] : result = 1.0
problem.state.current_best.x = [1 0 0 0 0]
```

Figure 2: Test of Leading Ones on two candidates with same fitness

## 2.7 Pseudocode

**Algorithm 1** CatSo_pop2_bias_rate (bias + backup with stochastic restarts)

**Require:** budget $B$, restart rate $r$, mean restart budget $\mu$, variance $\sigma^2$, bias rate $\beta$

Sample $x_f, x_b \sim \{0,1\}^n$ ▷ favorite and backup
Evaluate $f_f \leftarrow F(x_f)$, $f_b \leftarrow F(x_b)$; ensure $x_f$ is the best of the two
Set bit-flip rate $p \leftarrow 1/n$
**while** evaluations $< B$ and not optimum **do**
  **if** Uniform$(0,1) \leq r$ **then** ▷ restart episode
    $m \leftarrow \max\{0,\ \lfloor \mathcal{N}(\mu, \sigma^2) \rfloor\}$
    Sample $y \sim \{0,1\}^n$, evaluate $g \leftarrow F(y)$
    **for** $t = 1$ to $m$ **do**
      $y' \leftarrow$ bitwise-mutate$(y, p)$ (force at least one flip allowed)
      $g' \leftarrow F(y')$;
      **if** $g' \geq g$ **then** $y \leftarrow y'$, $g \leftarrow g'$
      **end if**
      **if** optimum found **then break**
      **end if**
    **end for**
    **if** $g \geq f_b$ **then**
      **if** $g \geq f_f$ **then**
        $x_b \leftarrow x_f,\ f_b \leftarrow f_f;\ x_f \leftarrow y,\ f_f \leftarrow g$
      **else**
        $x_b \leftarrow y,\ f_b \leftarrow g$
      **end if**
    **end if**
  **else** ▷ normal step on favorite or backup
    **if** Uniform$(0,1) \leq \beta$ **then** ▷ explore favorite
      $z \leftarrow$ bitwise-mutate$(x_f, p)$; $h \leftarrow F(z)$
      **if** $h \geq f_f$ **then** $x_f \leftarrow z,\ f_f \leftarrow h$
      **end if**
    **else** ▷ explore backup
      $z \leftarrow$ bitwise-mutate$(x_b, p)$; $h \leftarrow F(z)$
      **if** $h \geq f_b$ **then**
        **if** $h \geq f_f$ **then**
          $x_b \leftarrow x_f,\ x_f \leftarrow z;\ f_b \leftarrow f_f,\ f_f \leftarrow h$
        **else**
          $x_b \leftarrow z,\ f_b \leftarrow h$
        **end if**
      **end if**
    **end if**
  **end if**
**end while**

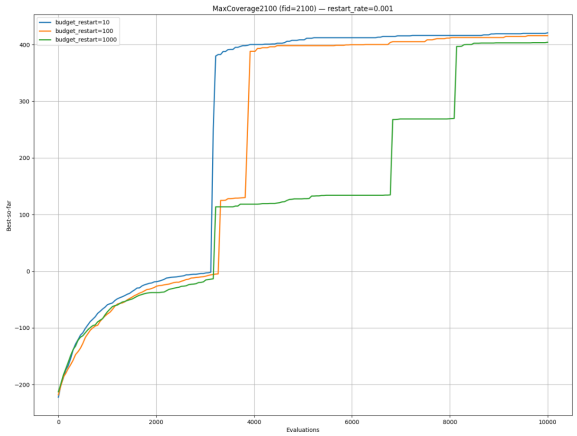


Figure 3: Comparisons of restart budgets on MaxCoverage

# 3 Results

## 3.1 Study 1: Restart rate and restart budget

We first implemented a simple version of our algorithm, **CatSo_restart_only**, which performs restarts with a fixed budget, without normal distribution and with a population size of 1 (no backup, no bias). We tested three restart rates:

$$\texttt{restart\_rate} \in \left\{ \frac{10}{B}, \frac{1}{B}, \frac{1}{10B} \right\},$$

and three restart budgets:

$$\texttt{budget\_restart} \in \left\{ \frac{B}{1000}, \frac{B}{100}, \frac{B}{10} \right\}.$$

The results showed that very frequent restarts with large budgets waste time, while too rare restarts and smaller budgets make it hard to escape local optima. (see Fig. 3) The most consistent performance across problem types was obtained for:

$$\boxed{\texttt{restart\_rate} = 10/B, \quad \texttt{budget\_restart} = B/10.}$$

This setting balances exploration and exploitation efficiently, giving enough time for each restart to prove itself useful before switching back.

## 3.2 Study 2: Distribution of restart budget

We then extended the algorithm to draw each restart budget from a normal distribution

$$N(\mu = \texttt{mean\_budget\_restart},\ \sigma^2 = \texttt{var\_budget\_restart}),$$

and investigated the effect of the variance. The mean was fixed to 100, and we tried variances

$$\sigma \in \{\mu/10,\ \mu/50,\ \mu/100\}.$$

On almost all tests we ran, the curves were very similar regardless of variance (see Fig. 4). This indicates that the precise value of the variance does not significantly affect performance. Consequently, we decided to fix $\sigma = \mu/50 = 20$ when the budget is $10,000$ for subsequent experiments.

## 3.3 Study 3: Bias rate in two-population variant

Next, we implemented **CatSo_pop2_bias_rate**, a two-population variant maintaining a *favorite* and a *backup* candidate. The algorithm chooses which one to mutate according to a bias rate $b$: with probability $b$ it mutates the favorite, and otherwise the backup. If a child from the backup outperforms the favorite, the two swap roles.

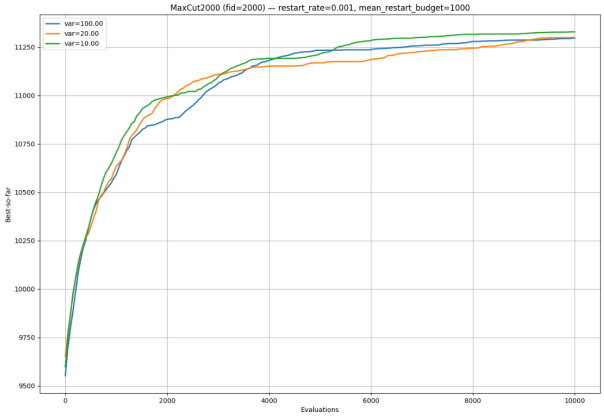


Figure 4: Variance difference on MaxCut

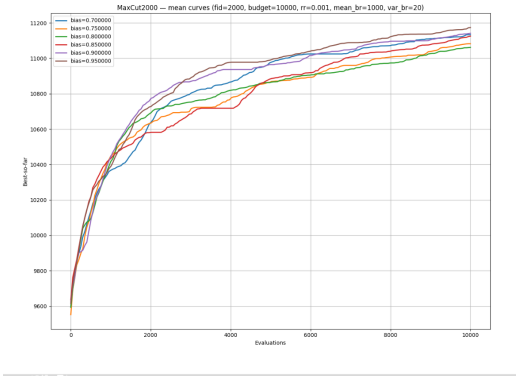


Figure 5: Comparisons on bias rates in MaxCut

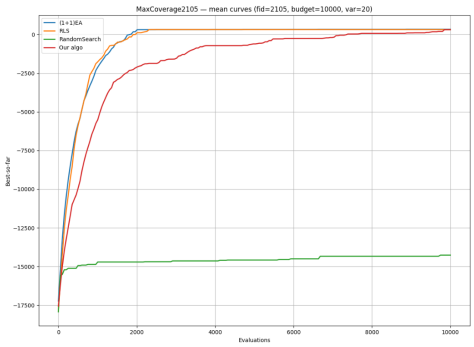


Figure 6: Classical algorithms & ours in MaxCoverage

We fixed the restart settings from the previous experiment (`restart_rate` $= 1/B$, `mean_budget_restart` $= 100$, `var_budget_restart` $= 20$) and varied

$$\texttt{bias\_rate} \in \left\{1 - \frac{10}{B}, 1 - \frac{50}{B}, 1 - \frac{100}{B}, 1 - \frac{500}{B}, 1 - \frac{1000}{B}\right\}.$$

As expected, very high bias rates (close to 1) favor faster exploitation (see Fig. 5), while smaller ones allow more exploration through the backup candidate. On graph problems with many local optima, such as `MaxInfluence`, intermediate values around $0.9-0.95$ performed best.

### 3.4 Study 4: Comparison with classical algorithms

We finally compared our best-performing version (CatSo with restarts and bias) against Random Search, RLS, and the $(1+1)$ EA across several problem instances (see Fig. 6).

In `MaxInfluence` and `PackWhileTravel`, the $(1+1)$ EA still achieves faster convergence overall, but our algorithm follows closely, and even occasionally overtake them in the problems (`MaxCut`, `MaxCoverage`). RLS performed the worst out of all algorithms put to the test, in all instances.

## 4 Discussion

Our results confirm that restarts can substantially improve exploration in multimodal search spaces but need careful tuning. The restart rate controls how often we give up exploitation, and the restart budget determines how much time each new exploration path receives. Adding a second population axis introduces robustness: even if a restart fails to outperform the favorite, it can still replace a weaker backup and be revisited later (see Fig. 9). The bias parameter offers a smooth trade-off between aggressive exploitation (high bias) and safer exploration (lower bias). Interestingly, the variance of restart budgets had little impact—suggesting that most

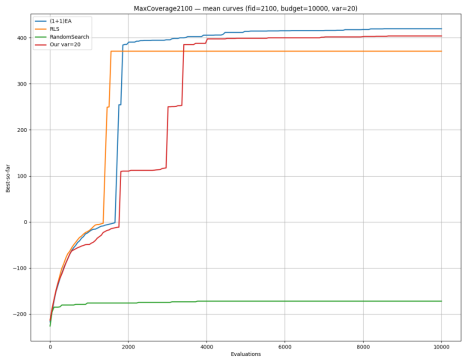


Figure 7: Comparisons with variance on MaxCoverage

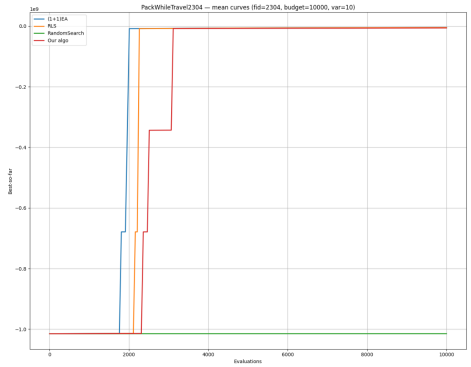


Figure 8: Comparisons with variance on PackWhileTravel

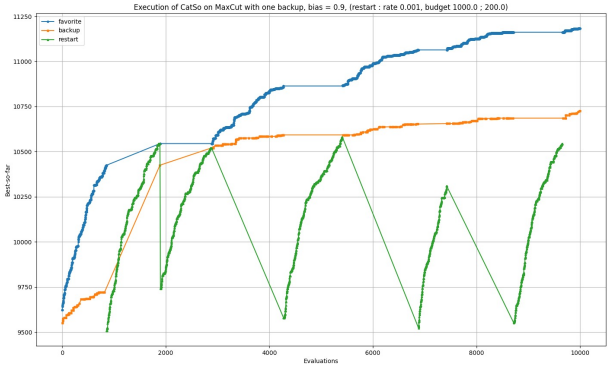


Figure 9: Evolution of the favorite, the backup, and the restarts.

of the improvement comes from controlling the mean and restart frequency rather than stochastic diversity in the restart duration.

Figure 9 shows the evolution of the

# 5 Complexity analysis

Each evaluation of a candidate flips bits independently and scans the full vector of length $n$, hence the *per–evaluation* time is $O(n)$. Let $B$ be the global evaluation budget, $r$ the restart probability per decision, and let the restart sub–budget be a random variable $M$ with $E[M] = \mu$ (in our experiments $\mu = \texttt{budget}/10$ and $\mathrm{Var}(M) = \sigma^2$).

At each decision we either perform one normal mutation (cost 1 evaluation) with probability $1 - r$, or we trigger a restart episode with probability $r$ whose expected cost is $1 + \mu$ evaluations (one new random point plus $\mu$ local steps). Therefore the expected cost per decision is

$$E[\text{cost per decision}] = (1 - r) \cdot 1 + r \cdot (1 + \mu) = 1 + r\mu.$$

Under a total budget $B$, the expected number of decisions is

$$E[\#\text{decisions}] \approx \frac{B}{1 + r\mu},$$

the expected number of restart episodes is

$$E[\#\text{restarts}] \approx \frac{rB}{1 + r\mu},$$

and the expected number of normal mutation steps is

$$E[\#\text{mut-steps}] \approx \frac{(1 - r)B}{1 + r\mu}.$$

Overall time is linear in $B$ with an $O(n)$ factor per evaluation:

$$T(B, n) = \Theta\big(n\, B\big),$$

and restarts only change the constant by the factor $1/(1 + r\mu)$ in the number of *decisions*, not the asymptotic complexity.

# 6 Conclusion

Our exploration started from the hypothesis that *where* we spend the budget matters as much as *how* we mutate. The bias + backup design explicitly encodes this belief: we keep two incumbents (a favorite and a backup) and we decide, with probability $\beta$, to push the favorite forward, otherwise we invest in the backup. Across MaxCut, MaxCoverage and PackWhileTravel we consistently observed that a *moderate* restart rate ($r = \frac{1}{\text{budget}}$) with a *chunky* local budget inside the restart ($\mu = \frac{\text{budget}}{10}$) is a robust setting. Variance of the sub-budget had a much smaller effect than expected and we opted for a small fixed value. With $\beta = 0.9$ the favorite/backup mechanism yielded smooth gains without starving the backup, and recovered quickly from unlucky local moves. In short, the bias + backup idea gave us a practical way to allocate search effort dynamically: stay committed to the best incumbent most of the time, but keep a credible Plan B and give restarts enough runway to actually improve before we judge them.