Comparing Self-Adjusting \((1 + \lambda)\) EAs under Large Dimensions: A Case Study

Arina Buzdalova (ITMO) Carola Doerr (Sorbonne) Anna Rodionova (ITMO) Kirill Antonov (ITMO)

Working Group Workshop
COST Action CA15140
February 19, 2019
Experiment description

- $(1 + \lambda)$ EA with “practice-aware” shift mutation
- 4 self-adjusting $(1 + \lambda)$ EAs:
  - 2 different rules for updating mutation rate:
    - 2-rate: asymptotically optimal runtime for large enough $\lambda$ (theoretically proven)
    - Ab: chooses number of bits close to optimal (empirically shown)
  - 2 different mutation lower bounds: $1/n$ and $1/n^2$
- Problem: OneMax
- Dimensions
  - problem size $n = 10000, 20000, \ldots 100000$
  - population size $\lambda = 1, 5, 10, 50, 100, 200, 400, 800, 1600, 3200$
- 100 independent runs of each algorithm
Compared algorithms: \((1 + \lambda) \text{ EA}_{0 \rightarrow 1}\)

**Algorithm 1:** The \((1 + \lambda) \text{ EA}_{0 \rightarrow 1}\) with mutation rate \(p \in (0, 1)\) for the maximization of \(f : \{0, 1\}^n \rightarrow \mathbb{R}\)

1. **Initialization:** Sample \(x \in \{0, 1\}^n\) u.a.r.;
2. **Optimization:** for \(t = 1, 2, 3, \ldots\) do
   3. for \(i = 1, \ldots, \lambda\) do
      4. Sample \(\ell^{(i)}\) from \(\text{Bin}_{0 \rightarrow 1}(n, p)\), sample \(y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)\) and evaluate \(f(y^{(i)})\);
   5. Sample \(x^*\) from \(\arg\max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}\) u.a.r.;
   6. if \(f(x^*) \geq f(x)\) then \(x \leftarrow x^*\);
Compared algorithms: 2-rate \((1 + \lambda) \text{EA}_{r/2,2r}\)

---

**Algorithm 2:** The 2-rate \((1 + \lambda) \text{EA}_{r/2,2r}\) with adaptive mutation rates proposed in [DoerrGWY17]

1. **Initialization:** Sample \(x \in \{0, 1\}^n\) uniformly at random and evaluate \(f(x)\);
2. Initialize \(r \leftarrow r^{\text{init}}; \quad \text{// We use } r^{\text{init}} = 2;\)
3. **Optimization:** for \(t = 1, 2, 3, \ldots\) do
   - for \(i = 1, \ldots, \lfloor \lambda/2 \rfloor\) do
     - Sample \(\ell(i) \sim \text{Bin}_{0 \rightarrow 1}(n, r/(2n))\), create \(y(i) \leftarrow \text{flip}_{\ell(i)}(x)\), and evaluate \(f(y(i))\);
   - for \(i = \lfloor \lambda/2 \rfloor + 1, \ldots, \lambda\) do
     - Sample \(\ell(i) \sim \text{Bin}_{0 \rightarrow 1}(n, 2r/n)\), create \(y(i) \leftarrow \text{flip}_{\ell(i)}(x)\), and evaluate \(f(y(i))\);
4. \(x^* \leftarrow \arg \max\{f(y(1)), \ldots, f(y(\lambda))\}\) (ties broken u.a.r.);
5. if \(f(x^*) \geq f(x)\) then \(x \leftarrow x^*;\)
6. Perform one of the following two actions with prob. 1/2:
   - replace \(r\) with the mutation rate that \(x^*\) has been created with;
   - replace \(r\) with either \(2r\) or \(r/2\) equiprobably.
7. \(r \leftarrow \min\{\max\{2, r\}, n/4\};\)
Compared algorithms: \((1 + \lambda)\) EA\((A, b)\)

**Algorithm 3:** The \((1 + \lambda)\) EA\((A, b)\) with adaptive mutation rates and update strengths \(A > 1, 0 < b < 1\)

1. **Initialization:** Sample \(x \in \{0, 1\}^n\) uniformly at random and evaluate \(f(x)\);
2. Initialize \(p \leftarrow 1/n\); **Optimization:** for \(t = 1, 2, 3, \ldots\) do
   3. for \(i = 1, \ldots, \lambda\) do
      4. Sample \(\ell(i) \sim \text{Bin}_{0 \rightarrow 1}(n, p)\), create \(y(i) \leftarrow \text{flip}_{\ell(i)}(x)\), and evaluate \(f(y(i))\);
      5. \(N \leftarrow |\{i \in [\lambda] \mid f(x(i)) \geq f(x)\}|\);
   6. if \(N \geq \lceil 0.05\lambda \rceil\) then \(p \leftarrow \min\{1/2, Ap\}\) else \(p \leftarrow \max\{1/n, bp\}\);
   7. \(x^* \leftarrow \text{arg max}\{f(x(1)), \ldots, f(x(\lambda))\}\) (ties broken u.a.r.);
   8. if \(f(x^*) \geq f(x)\) then \(x \leftarrow x^*\);
Comparison regarding different population sizes

Number of generations:

- Almost opposite ranking for small and large population sizes $\lambda$
- For both algorithms, $1/n^2$ lower bound is better for small $\lambda$, $1/n$ is better for large $\lambda$
Comparison regarding different population sizes

Number of fitness evaluations:
Comparison regarding different population sizes

Number of generations averaged by 100 runs and its standard deviation. Shift mutation operator is used in all algorithms, avg.=average, r.dev.=relative standard deviation.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>$(1 + \lambda) \text{ EA}_{0\rightarrow 1}$</th>
<th>2-rate $(1/n)$</th>
<th>Ab $(1/n)$</th>
<th>2-rate $(1/n^2)$</th>
<th>Ab $(1/n^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg.</td>
<td>r.dev.</td>
<td>avg.</td>
<td>r.dev.</td>
<td>avg.</td>
</tr>
<tr>
<td>$n = 100000$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1,873,666</td>
<td>13.4%</td>
<td>2,222,691</td>
<td>10.9%</td>
<td>1,860,718</td>
</tr>
<tr>
<td>5</td>
<td>389,110</td>
<td>10.2%</td>
<td>716,372</td>
<td>13.6%</td>
<td>420,165</td>
</tr>
<tr>
<td>10</td>
<td>205,086</td>
<td>12.0%</td>
<td>414,043</td>
<td>10.4%</td>
<td>248,184</td>
</tr>
<tr>
<td>50</td>
<td>60,200</td>
<td>6.2%</td>
<td>98,223</td>
<td>8.7%</td>
<td>76,772</td>
</tr>
<tr>
<td>100</td>
<td>42,151</td>
<td>5.0%</td>
<td>57,325</td>
<td>6.9%</td>
<td>41,520</td>
</tr>
<tr>
<td>500</td>
<td>31,762</td>
<td>3.4%</td>
<td>37,588</td>
<td>6.8%</td>
<td>28,215</td>
</tr>
<tr>
<td>1000</td>
<td>25,846</td>
<td>2.1%</td>
<td>26,227</td>
<td>5.0%</td>
<td>20,900</td>
</tr>
<tr>
<td>800</td>
<td>22,229</td>
<td>1.2%</td>
<td>20,055</td>
<td>3.2%</td>
<td>16,681</td>
</tr>
<tr>
<td>1,600</td>
<td>19,744</td>
<td>0.7%</td>
<td>16,325</td>
<td>2.0%</td>
<td>14,112</td>
</tr>
<tr>
<td>3,200</td>
<td>17,956</td>
<td>0.4%</td>
<td>13,966</td>
<td>1.1%</td>
<td>12,296</td>
</tr>
<tr>
<td>$n = 10000$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>147,008</td>
<td>14.8%</td>
<td>177,568</td>
<td>14.8%</td>
<td>148,182</td>
</tr>
<tr>
<td>5</td>
<td>31,738</td>
<td>17.5%</td>
<td>56,940</td>
<td>17.3%</td>
<td>34,514</td>
</tr>
<tr>
<td>10</td>
<td>16,373</td>
<td>13.1%</td>
<td>32,054</td>
<td>15.3%</td>
<td>20,662</td>
</tr>
<tr>
<td>50</td>
<td>5,254</td>
<td>7.5%</td>
<td>8,029</td>
<td>12.9%</td>
<td>6,855</td>
</tr>
<tr>
<td>100</td>
<td>3,790</td>
<td>4.6%</td>
<td>4,922</td>
<td>9.4%</td>
<td>3,662</td>
</tr>
<tr>
<td>200</td>
<td>2,955</td>
<td>3.6%</td>
<td>3,323</td>
<td>8.2%</td>
<td>2,647</td>
</tr>
<tr>
<td>400</td>
<td>2,486</td>
<td>2.1%</td>
<td>2,420</td>
<td>5.7%</td>
<td>2,001</td>
</tr>
<tr>
<td>800</td>
<td>2,164</td>
<td>1.1%</td>
<td>1,903</td>
<td>3.8%</td>
<td>1,633</td>
</tr>
<tr>
<td>1,600</td>
<td>1,945</td>
<td>0.8%</td>
<td>1,582</td>
<td>2.1%</td>
<td>1,393</td>
</tr>
<tr>
<td>3,200</td>
<td>1,776</td>
<td>0.6%</td>
<td>1,379</td>
<td>1.8%</td>
<td>1,227</td>
</tr>
</tbody>
</table>
Comparison regarding different problem sizes
Fixed Budget Results: small population size $\lambda = 10$
Fixed Budget Results: medium population size $\lambda = 400$
Fixed Budget Results: large population size $\lambda = 1600$
Fast implementation of the standard mutation operator

calc_fitness (patch, best_individual, best_fitness, mask) {
    patch_fitness := best_fitness
    for (i in patch){
        if (best_individual[i] == mask[i])
            patch_fitness := patch_fitness - 1
        else
            patch_fitness := patch_fitness + 1
    }
    return patch_fitness
}

apply_patch(patch, best_individual) {
    for (i in patch)
        best_individual[i] := 1 - best_individual[i]
Conclusion

- (1 + \(\lambda\)) EA and 4 self-adjusting (1 + \(\lambda\)) EAs were compared on OneMax for \(n = 10^4 \ldots 10^5, \lambda = 2 \ldots 3200\)
- Results strongly influenced by the population size:
  1. Ranking in terms of runtime
  2. Ranking in terms of fixed budget
  3. Relative standard deviation
- (1), (3) does not depend so much on problem size
- Possible consequences for benchmarking:
  - In a benchmarking framework, plotting against wide parameter range should be available
  - Fast implementation of fitness evaluation and standard operators may be needed