Improving Fuzzing performance with Evolutionary Computing

Carlo Baylon, Chidambaram Chidambaram, Thomas Stanton

Table of Contents

[**Background**](#_ogdd3tqpxysw) **3**

[Nosj and Implementations](#_j33w3zyzcxld) 3

[**Design**](#_3znysh7) **3**

[Evolutionary Algorithm: Overview](#_9jh3uad88r2h) 3

[Initial Population](#_8qcsjdthmujq) 4 [4](#_eny5p2j5a3pt)

[Evaluation/Fitness Function](#_fhkxy3tsj2gu) 5

[Parent Selection](#_mxcoix80nvmx) 6

[Recombination](#_gp60a5y1l1co) 6

[Mutation](#_ldvrgmji4mkg) 7

[Survival Selection](#_f4rmzh6fgd10) 7

[Convergence](#_ig9j6opf3s60) 7

[**Experiments**](#_2s8eyo1) **8**

[Process](#_raqri7vznbvx) 8

[Configuration](#_rvaz95495u59) 9

[**Results**](#_kj0pymyffnu9) **9**

[**Discussion**](#_wwl1hfl1oam1) **11**

[Fitness value comparison](#_bwq7ncvcqzon) 12

[String sets](#_ghekchqc26r9) 14

[Future improvements](#_bvu8o3nej1tg) 19

[**Appendix A**](#_pn5d8q907418) **20**

[Random Search](#_mhfhxrqqd2cl) 20

[**Hill Climb**](#_1t3h5sf) **20**

[**Appendix B**](#_zgnx4rn06yk) **21**

# 

# **Background**

Untrusted input(s) is one of the most common weaknesses found across software applications. It is essential to test for this weakness before an application can be deployed. Fuzzing is an automated testing technique where a program generates semi-random inputs to an application to check for weaknesses like untrusted inputs. In this paper we look at fuzzing and how to use AI techniques, in this case Evolutionary Algorithms, to improve fuzzing performance.

### Nosj and Implementations

For the purposes of testing our Evolutionary Algorithm we are using a data structure called NosJ. More specifically we have taken a number of different implementations of the NosJ standard that all seek to unmarshal the data structure into a Python dictionary. Each of these NosJ implementations take a set of strings and those string inputs are what we will be fuzzing.

We explore fuzzing nosj implementations with these two objectives.

1. Maximize the number of implementations that crash due to one or more of the fuzzing strings
2. Maximize the number of exception types raised when breaking these implementations

# 

# 

# 

# **Design**

We use an Evolutionary Algorithm to improve fuzzing performance and we compare results with Random Search and Hill Climb techniques (See Appendix A for details of Random Search and Hill Climb)

## **Evolutionary Algorithm: Overview**

Our Evolutionary Algorithm(or EA) follows the standard procedure of most EAs. Our generation cycle follows figure 1 and goes as follows:

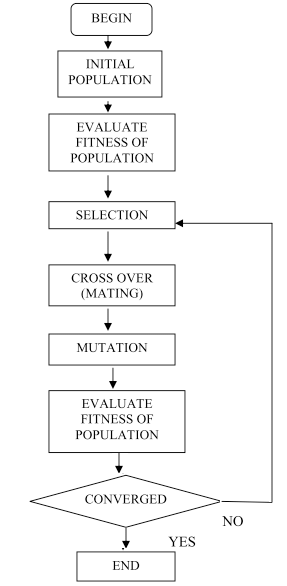


Figure 1: Our EA Cycle

First, an initial population of fuzzer sets are generated along with the evaluation of fitness of each individual set. Afterwards, we select parents based on their fitness in order to generate children based on the selected parent’s genes and add the children to the population. We chose to keep the population a consistent number, so after the children are added we select who survives from the population and remove the fuzzing sets that do not survive in order to do so. Finally, we test if our convergence criteria are met at the end of the generation cycle, and if not we start a new generation by looping back to the selection phase. If convergence is met, the algorithm concludes and has a resulting population with the best optimized fuzzing sets for our objective.

## **Initial Population**

Our initial population is initialized with a predefined number of fuzzing sets with randomly generated genomes. This is our first population in its first generation.

### ***Fuzzing set characteristics***

Our population consists of individuals we chose to define as fuzzing sets. Fuzzing sets contain a genome and their individual fitness.

The genome contains two parts: a set of strings as well as the length of the set. We chose the genome this way because evolution will help optimize both the values of these strings through mutations and the number of strings in the set through recombination. The fitness for each set is how well the set of strings in the genome perform on the mentioned implementations.

## **Evaluation/Fitness Function**

### ***Fitness evaluation flow***

For each implementation we send randomly generated strings (50 in our experiments) and record the exceptions thrown for each of the implementations(Figure 2). From the resulting matrix we calculate the fitness

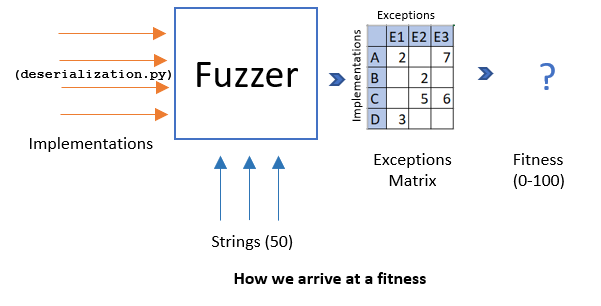


Figure 2: Process of String Set Evaluation

The exception matrix contains the number of exceptions seen (for each exception class) in an implementation. In the following example(Figure 3) implementation B saw ‘2’ exceptions of type E2.

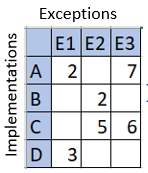


Figure 3: Example fitness matrix

We derive two fitness values out of this matrix ‘Exception Fitness’ and ‘Implementation Fitness’.

### ***Exception Fitness***

The exception fitness value resents the average percentage of exception classes raised per implementation. Calculated as

### ***Implementation Fitness***

The implementation fitness value represents the percentage of implementations in which at least one exception class was raised. Calculated as

### ***Weightage***

The two fitness values are then multiplied by two weights. ‘Exception Weight’ and ‘Implementation weight’ which represents how much a user cares about number of exception classes seen and number of implementations broken respectively. The program uses alpha to represent ‘Exception Weight’ and 1-alpha for ‘Implementation weight’. Represented by the following formula.

**+**

## **Parent Selection**

Once a population has been rated by our evaluation function, we proceed to parent selection. Here we choose ‘n’ individuals from the population using a K-Tournament with Replacement algorithm with a ‘k’ of 5. In general terms, this algorithm holds ‘n’ tournaments each with ‘k’ individuals randomly chosen from the population. The top fitness individual wins each tournament and is chosen to be a parent. This algorithm was chosen for its ability to favor high fitness individuals in the population while still allowing for some lower fitness parents.

## **Recombination**

The recombination follows parent selection. Once the parents are selected from the population, we select two parents to create a child. We chose two because having two parents is the smallest amount needed for recombination, while having more than two runs the risk of losing diversity in the population. We create children from the two parents with the following genome specifications:

### ***Child Genome: String set length***

The child’s string set length is sampled from a random distribution, with the mean being the average of the two parent’s string set lengths (μ = (p1\_length + p2\_length) /2) and a standard deviation σ = 1. This ensures that the child has a small chance of either having more or less strings in its genome set than its parents in order for our evolution to optimize the most efficient number of strings per set when testing against implementations.

### ***Child Genome: String set contents***

The child’s string set is a combination of genes randomly sampled from a set of both string sets of its parents. The sampling process makes sure to not use more than one string from either parent in the case that both parents contain the same string in their genomes to ensure that the child will not have duplicate strings. This is because duplicate strings will cause the same crashes on the same implementations such that a duplicate string will only serve to increase the set string’s length but not the fitness.

## **Mutation**

When a child is made after recombination, there is a chance that the genome of the child is mutated before being put in the population. This is to ensure that the population does not become homogeneous, while also allowing the population to gradually explore the entire fuzzing set space.

### ***Mutation definition***

Mutations directly preceded our neighbor definition(Appendix B1). In summary, a neighbor with the exact same fuzzing set except one string in the fuzzing set’s genome’s key or value is changed. As with our string analysis (Appendix B1), loosely valid nosj was prone to throwing more errors compared to both completely valid nosj and arbitrary strings of random characters.

A mutation is defined as the following with examples:

*Format: Original String -> Mutated string (mutation changes in red)*

* Changing the ordering of the colon, closing brace and opening brace : {abc:123} -> ab{c1:23}
* Omission of the colon, closing brace, and opening brace: {abc:123} -> |abc:123}
* Adding special characters and invalids in the key or value: {abc:123} ->{ab%J//5c:123}
* If no braces or brackets are found, addition of random characters of random length are added in a random index of the string: abc123 -> ab%/22s/5c123

Mutations are formatted in a way to have more bias towards valid nosj, but it is always feasible for the mutation to theoretically both omit every nosj format and purely add from the set of valid and invalid characters into the fuzzing set, allowing the string to be reformatted in any possible way in order for mutations to search for the global optimum in the case that the fuzzing set that contains the global optimum has strings with no valid nosj.

## **Survival Selection**

After we have finished creating and mutating children, it is time to add them to the population with their parents and trim it back down to its original size. For this, we choose a K-Tournament without Replacement algorithm with a ‘k’ of 5. In general terms, this algorithm holds ‘n’ (in this case ‘n’ is the size of our initial population to combat population growth)tournaments each with ‘k’ individuals randomly chosen from the population. The top fitness individual wins each tournament and is chosen to survive to the next generation. This algorithm was chosen for its ability to favor high fitness individuals in the population while still allowing for some lower fitness survivors.

## 

## 

## **Convergence**

Convergence or condition of convergence is required to terminate the search. If this criteria is not met, then the EA cycle will repeat again which we call a new generation. This process will repeat until the population has converged. For convergence we looked at two data points:

1. The best fitness individual’s distance from the average population’s fitness

This is the difference between the ‘best fitness’ individual in the population and the ‘average fitness’ of all individuals in the population. This will tell us if the population is relatively homogeneous.

1. Population’s average fitness distance from the median



This is the average difference between the median fitness of the population and the average fitness distance to median (ie. average of distance between each fitness to the median fitness). This will account for outliers when determining homogeneity.

This distance parameter (criteria for convergence) is configurable, in our experiments we used a value of 2% for both conditions above.

# 

# 

# 

# 

# **Experiments**

## **Process**

Our experiment(figure 4) consisted of running our EA 30 times to collect statistically significant amounts of data. Each time we ran our EA we collected three data points: the best fitness as a function of time log, the best fitness seen during the run, and the generation the run ended at. After repeating this process 30 times we were able to compile a “Best Fitness for All Runs” graph which shows us very general trends as well as what generation every run ended at. Next we take the highest performing run and use that data to create a “Best Fitness for Best Run” graph showing us information about how we found the Best performant strings.

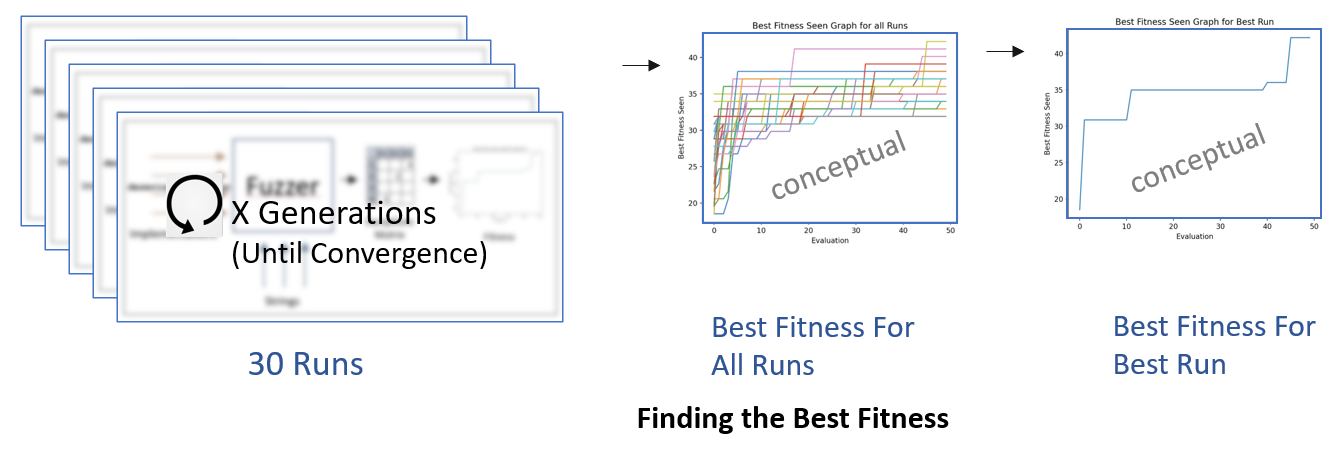


Figure 4: Process of data collection for experiment

Finally we take the Best Fitness for All Runs data and create a box and whisker plot to show how the EA ran on average at a given generation. One important thing to note for this graph is that since every run ended at a different generation, it was necessary to extrapolate the run’s final fitness value at convergence to match longer generations. Since each run ended only when it had converged, it can be assumed that there would be no meaningful increase in fitness in any future generation thus we deemed this acceptable to use for a graph.

## **Configuration**

An entire experiment run could easily be configured through configuration variables in our implementation, and initially we arbitrarily set variables such as mutation rate and first generation population size, but through testing we came up with an optimized set of configuration variables in respect to max amount of fitness gains and resources constraints.

We ran our experiments with a population size of 50 individuals with 20 children generated each generation. These two values were chosen due to resource constraints.

The starting population is generated with 10 strings each but each child has the possibility to be generated with more strings than one parent has. This growth is capped at 50 strings per genome and we chose to start at 10 and work our way to 50 to incentivise individuals finding high fitness solutions with minimal amounts of strings.

Each child generated had a 15% for each gene in their genome to contain a single mutation. We chose 15% after testing because any lower hampered genetic diversity and ended with either populations all looking alike or containing many duplicate strings. We choose not to go any higher for fears of adding too much randomness into the population each generation. The mutation rate of 15% is a necessity because it also helps compensate for such a low initial population.

The general fitness weight of alpha was given a value of 0.50, so a 50:50 split between implementations broken and exceptions causes. We chose this value because it ensured the vast majority of implementations were broken while still incentivising the casing of as many exceptions as possible.

# **Results**

Looking at the best run of our experiment in figure 5, we see a decently smooth and consistent increase in best fitness from generation 0 through 20. After this point we see a slow down in gains until around generation 40 where no more best fitness gains are made. Figure 6 shows that this explosive growth followed by a gradual leveling out of gains is a common trend in this EA. We believe the explanation of this trend is that in generations 0 through 20, fitness gains are mostly by recombination of individuals with diverse genetic material from the initial population. But after around generation 20 in a given population, the variance in the population's genes starts to decrease as the population converges.

This means that most of the new fitness gains are now caused by mutations rather than crossover. This is best demonstrated in figure 7 where we see the Best Fitness for All Runs from an earlier test with a mutation rate of 5% instead of the normal 15%. Here we see that rapid fitness gains still happen between generations 0 through 20 but after that not much gains are made, the populations rapidly converge, and the runs end around 10 generations earlier than in figure 6. This is because the experiment with a mutation rate of 5% does not have the high amount of mutations to drive further exploration of the search space after their populations have lost their genetic diversity.

Figure 8 shows how consistent each run of the EA was as well as how the EA reaches a relatively stable average population fitness around generation 26 on average.

Finally the set of best strings(see figure 9 EA Strings) was only of length 37 telling us that while 50 strings was the max, the algorithm felt no need to use all 50. While a high fitness with less than the maximum number of strings is desired, our fitness function had no penalty for more or less strings. This leads us to believe that our EA might not be aware of possible fitness to be gained by having more unique strings. Possible solutions to this could be increasing the number of strings per individual in the initial population or adding more individuals to the starting population. That being said, the 37 strings are all individually unique and contain a combination of maps, percent encoded characters, and regular alphanumeric characters. These strings gave us a best max fitness seen of 80.46296296296298.

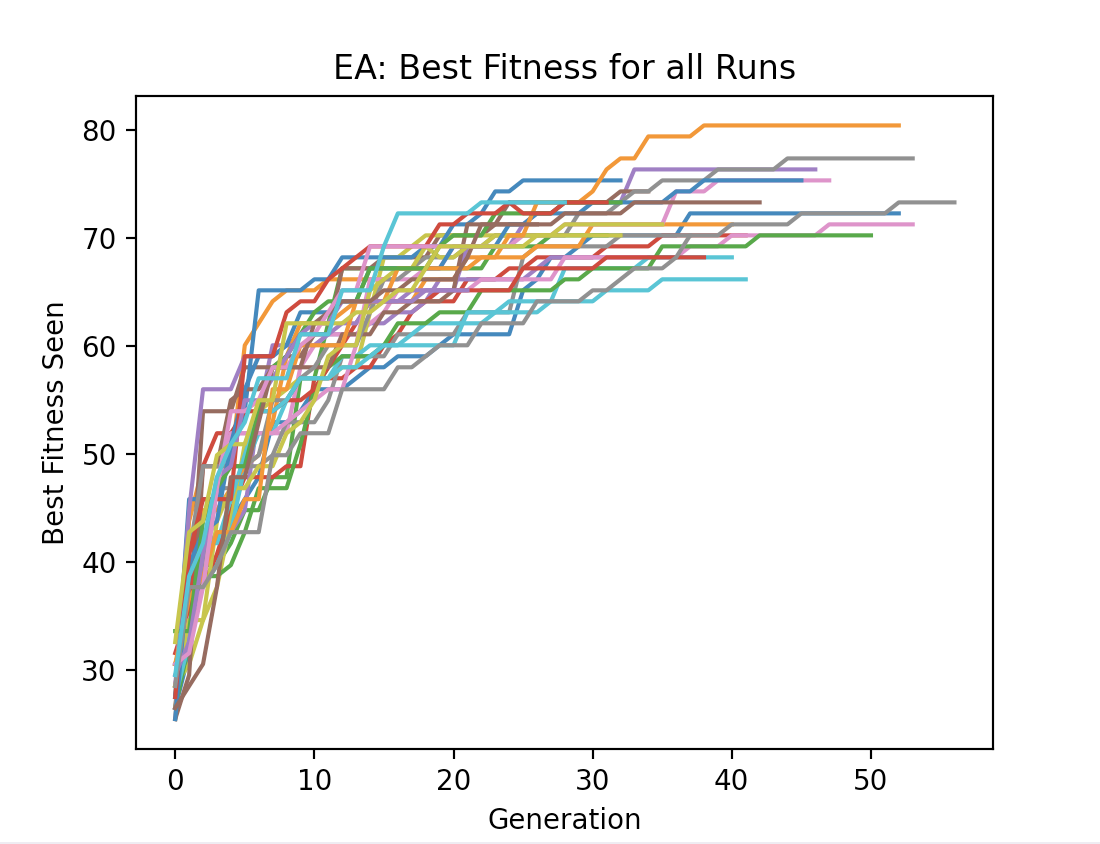
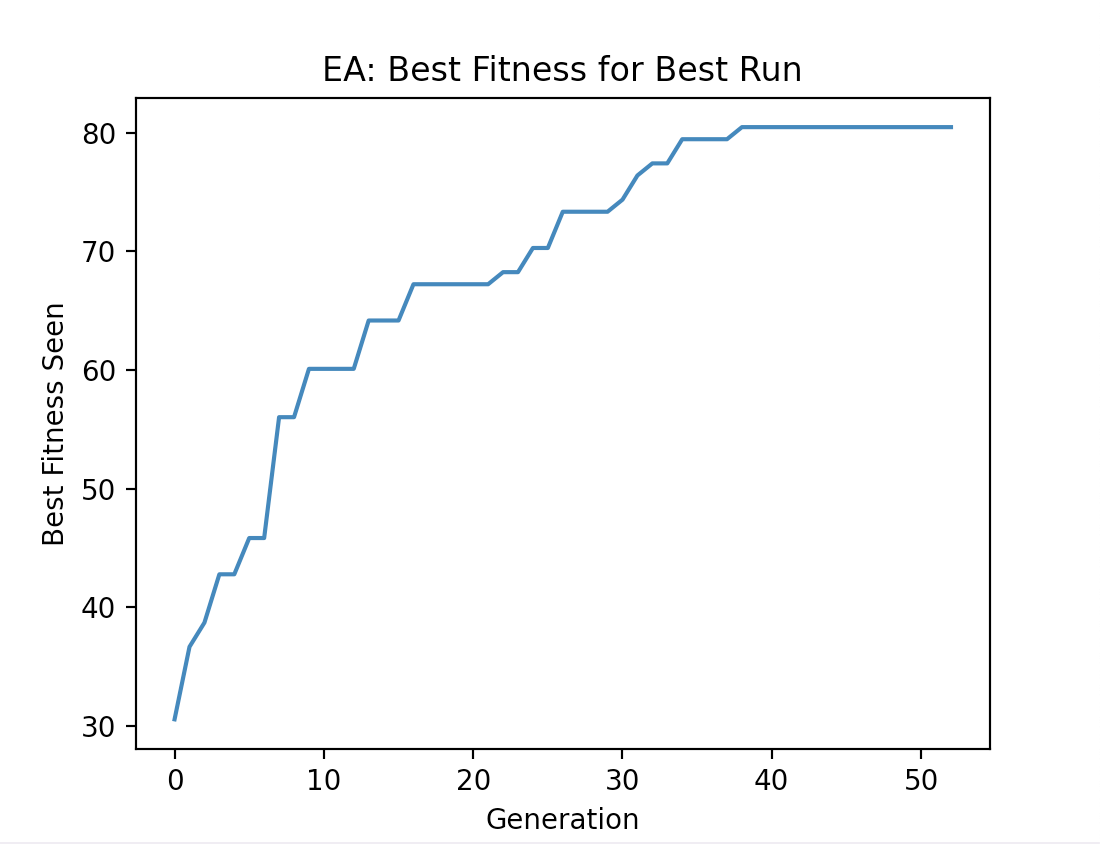


Figure 5: EA Best Fitness of Best Run Figure 6: EA Best Fitness for All Runs

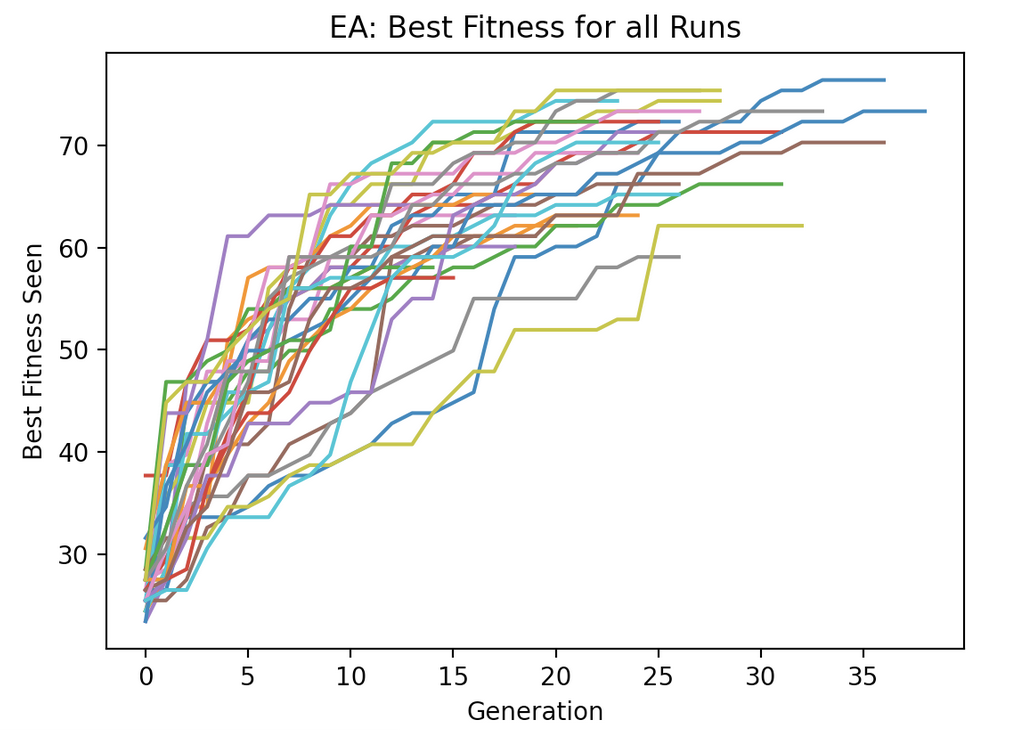
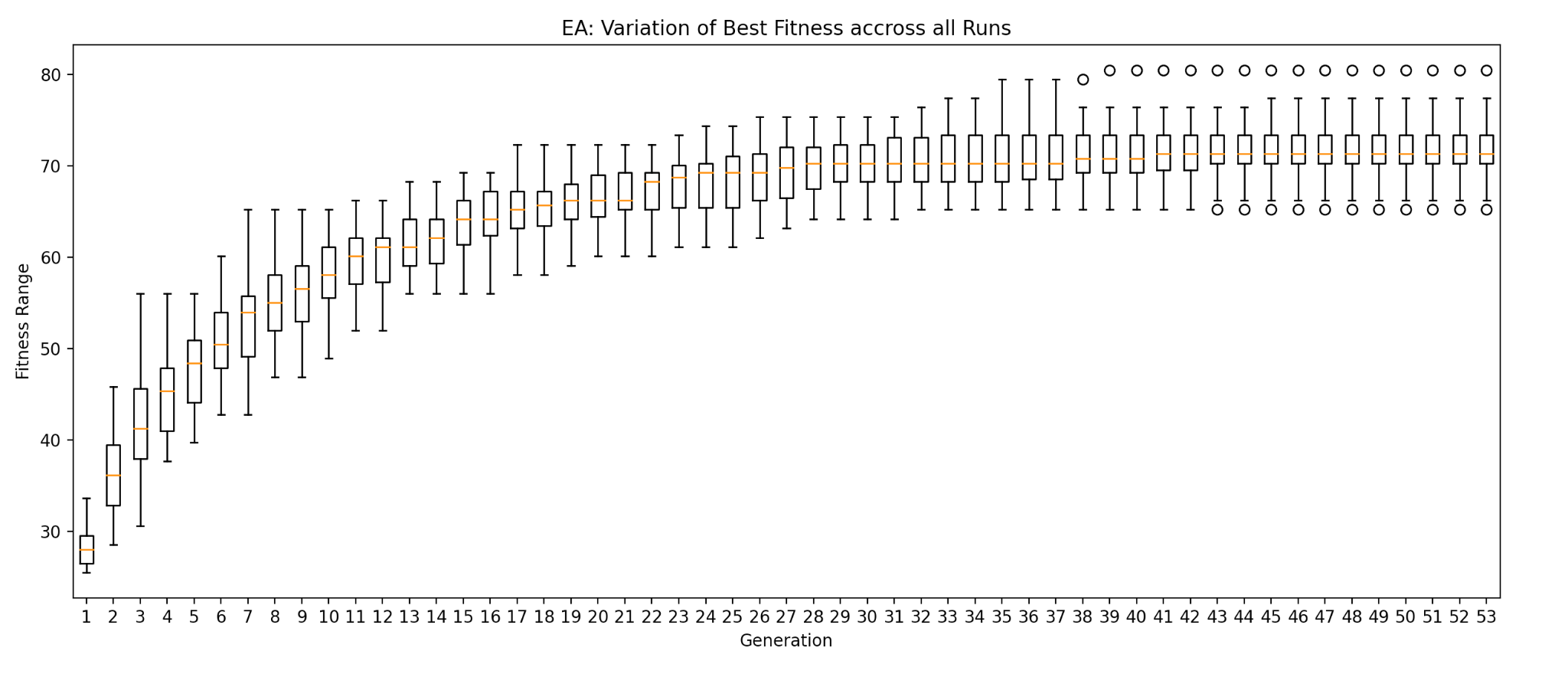


Figure 7: Earlier EA Best Fitness for all Runs(Mutation Rate = 5%)

Figure 8: EA Variation of Best Fitness across All Runs

# 

# 

# 

# 

# 

# 

# **Discussion**

Looking at figure 9 we see that when compared to a Random Search string generator and a Hill Climb string generator, our EA implementation found a much better performant string set. In figure 15 we see that the Hill Climb algorithm found string sets with fitness values ranging from mid 40s to low 60s. We also see that in figure 18, Random Search found values from the low 30s to mid 40s. Both found values much lower than the EA which had values ranging from low 60s to low 80s. Since all 3 algorithms were run with the same fitness function and same configuration variables where applicable, it can be assumed visusually that the EA out performed both Random Search and Hill Climb.

Moving on to the best performing strings themselves we see that in figures 11 and 12 that both the Random Search and Hill Climb algorithms produced strings that contained mostly valid NosJ maps, a few duplicate strings, and little to no percent encoded characters. This is contrasted to figure 10 in which the EA’s where we see very unique utilizing some valid NosJ maps (but also using broken maps), no duplicate strings, and lots of percent encoded characters.

We can see that our recombination techniques to maximize fitness with the least amount of strings were efficient enough to produce sufficient results when compared to the two other algorithms. Compared to random search, we saw roughly a 71% increase in best fitness while simultaneously reducing the number of strings per by 26% in our best fuzzing sets. We also saw comparable results compared to our hill climb approach, with an almost 34% increase in best fitness and reducing the best fuzzing string’s total strings by also 26%.

## **Fitness value comparison**

| **Search Algorithm** | Highest Fitness Seen |
| --- | --- |
| **Random Search Best Fitness** | **46.85185185185185** |
| **Hill Climb Best Fitness** | **60.09259259259259** |
| **Evolutionary Algorithm Best Fitness** | **80.46296296296298** |

Figure 9: Results of 3 Different Search Algorithms

**String sets**

| **EA Strings: 37 Total Strings | Fitness = 80.46296296296298** |
| --- |
| 1. **{TP8xQr: /^#)=s,VbELMN7yrH:3\*%#;8609422i}**   **6)Q.Ef(v\i\*%22$%%5B\_ K:KRBCpF:afhe\_7-i{zps}}**   1. **vUhh:HmritZ834605{679i,DSLns}** 2. **{{&$:'{:5W"}X.]4;t\_"{sha--}:.s}** 3. **{'7** 4. **"\*K#{CTY1E8Y4L8211s}:}** 5. **T@\*%{6:SF91,D7H}40+-"`P8xQr: /^#)=s, %60|=^%%3D{-\+VbELMN7yrH38609%5E422i}:** 6. **{ 6)Q.Ef(v\i}\*%22$%%5B\_ K:{K%5E%`%3DRBCpF:afhe\_7-izps}** 7. **{wSaMTjPZiTDs:,71;[.\ -)?!s}**   **c%5D,xwgr{lvd38:\_% 18i=[%60{",A4bE^2s}**   1. **{Wo:i238222i,ONGFPOq6ajhKz%7B%5B[Fs}** 2. **{%26%60<?<&$:'{:5W"}X.]4;t\_"{sha--}:.s}** 3. **{<| %2A %5E%22|~%2C'{{\*/** 4. **=> :{40.-W9%25!>+!N3W:4[5 3-}s}** 5. **Wo:i238222i,ONG{FPOq6ajhKz%{A2hD%}:7B%5B[Fs}** 6. **{%^@>+Wo:i238222i,ONGFP}Oq6ajhKz%7B%5B[Fs** 7. **{Woi238222:i,ONGFPOq6ajhKz%7B%5B[Fs}** 8. **W%2B%7B&o:i}{238222i,ONGFPO:q6ajhKzFs** 9. **{<\*/**   **=> {40.-W9:N3W4[5 3-}s}**   1. **{c%5D,xwgrlvd:38\_% 18i,A4bE:^2s}** 2. **cxw>\* ~grlvd:38\_%{ 18i,A4bE:^2s}**   **6)Q.Ef(v\i\_ {K:KRBC#%5D,/{%29%%27pF:afhe\_7-izps}}**   1. **{W\*:%%5E(}"/~<%%oi:238222i,ONGFPO:q6ajhKz%7B%5B[Fs}** 2. **{TP8xQr: /^#)=s,VbELMN7yrH:3\*%#;860942%%7D!%27;%27}.%2C!{",2i}** 3. **cxwgrlvd38:v03fOpcFcm0oxcV{'m%!9h:}18i,A4bE^2s}** 4. **{ENQJ6B492549272i:,27061}375:l'Ss** 5. **{Wo:i238222i,ONGFPOq6aj}%3F%22\_-hKz%7B%5B[Fs** 6. **cxwgrlvd38:{{v03fOpcFcm0oxcV'm%!9h:}18i,A4bE^2s}** 7. **cxwgrlvd38:{18i,{/00rY :+Z}A4bE^2s}** 8. **{}**   **<\*/**  **={> 40.-W9N3W4[5 3:-}s}**   1. **{|=** 2. **#T=Q!}%?},w:{fUIQrlle:b} '%7B>jQeNnGs}}** 3. **JmT-TOoh:1719759i,gzv6l34:#s{}** 4. **{cxwgrlvd38:18i,A}4bE^2s}** 5. **{D ` /~%%24\,(^=s,XZ:3"%22{`\<%{~%2B21s}** 6. **{D ` /%%3D",=<|+<~%%24\,(^=s,XZ:3"%22{`\<%{~%2B21s}** 7. **{ 6)Q.Ef(v\i}\*%22$%%5B\_ K:{KRBCpF:afh% %7C[`} /=<>=e\_7-izps** 8. **wSaM:TjPZiTDs,71;[.\ -)?{!s}** 9. **{W^=.%5D^}&%2Boi:238222i,ONGFP%2C <O:q6a{M;4b/:d&LavBb** 10. **zFy4Wiz!m0Y7^U2gJJm\_vs4HWJDAF}jhKz%7B%5B[Fs}** |

Figure 10: Best Performant String Set from EA

| **Hill Climb Strings: 50 Total Strings | Fitness = 60.09259259259259** |
| --- |
| 1. **{nqC:+s,jHIVihtBDj::=\|-\_l:rzs}** 2. **{G:51s,uc:-' -`=s}** 3. **{70i,MU144:289525i}** 4. **{.:215i,uxFyDL:3i}** 5. **{3941792198:ps,b3gmef:194i}** 6. **{ne.murqi:Hbiis,JTmPZLBi:2Ls}** 7. **/.{\_**     1. **|(>[:{}|`2\_${i:{}}}** 8. **LoPXL:306798i,JB:Wj{vvqvs}** 9. **{62DOFWXJ[?%22%2A,|%7C%,7848608567:i,\_UkTs}** 10. **{qs:i$PljAMs,Z[\_1073:4426221i}** 11. **{\_V:#c>j!|X!Oms,BiQ:985i}** 12. **:{G}{}** 13. **{mORTrUo:dOigoVs,+xaq+\_>?$s}** 14. **{lzERDHICP:bths,k++smvbvc:1601479s}** 15. **{ir971i,gZ5807:767460i}** 16. **{km:375587635i,oDZng6666:$\_[^JDs}** 17. **{SbJq40UiG:489836i,8084:716s}** 18. **{\_"\*`$>:{akvZjRekD:XoBWS8dgns}}** 19. **{tpj:645260i,zujq42v:8926864i}** 20. **{AoALec:Hts,EazRTHDb:!\|~|s}** 21. **{^** 22. **:}}O.{^%][t:{$(s:{}}}** 23. **{FO:961340i,- \_:532047880s}** 24. **{}** 25. **{AXKUZ:Bs,73:,%22%.\*@lOxInVfs}** 26. **{omuya:'?[&'@[<.s,+TUXrGu:3092349s}** 27. **{NBWWDYQ:780i,Oxj:`|\_~:@4]!s}** 28. **{:~[}%29%28M}{}** 29. **{nvpHrdym:3cs,w:44917766i}** 30. **{oxvbxxm:70i,xUgjLaWZsE::{HWKDUe'</%}}** 31. **{\_\_-:94032808s,G:~s}** 32. **{io9RhzBj:@~38(@>4s,OPE:)#~\((s}** 33. **{;Z:{'g.k@~:j}.P\*.^C"j:{}}}** 34. **{}** 35. **{L9VL:9s,8FXW:ds}** 36. **{srDnzpx:[: 1]:s,IYMMAn:223739899s}** 37. **{QuY:945459221i,g q t+:8i}** 38. **{LKFaCvoTH:4).]48^s,FAWNpTcC:90-55<1]\s}** 39. **pnbJ:@/+{<#.'s,RIPG:.$ &~>s}** 40. **{{%}:~Q`8CZFP+:D}:Bms,VVnOBQOtv:71621061i}** 41. **{xofT0oO:42i,DPWLZMN:eBMyRJs}** 42. **{eKlX :8];(|\_4\*~s,C6M47H1:75884992i}** 43. **{@">:\_U B``h'! ,|U:{\_+-:Zs}}** 44. **{V:9(0@@'0=9~s,hBvebuDyUs:793644062i}** 45. **{YP.njbgMCY:96098828i,qwgoaMyX:js}:8}{{}** 46. **{pXcwrPDOzl:PU%3F,bs,PMGJFUBIS:/$ `-['s}** 47. **{|e,~)c:{},\_'/d>|<^]`\*b:{rK4lC:Bs}}** 48. **{UAvDIXC:bkqvnCs,U\_XUzI:28M5MZs}** 49. **{S:;@#s,wEL.tM:v?ux)r1'Ws}** 50. **{lfof:44Qan2v4V6s,\_7LP.eyQ:?`\*\*<\*=\*&$s}** |

Figure 11: Best Performant String Set from Hill Climb

| **Random Search: 50 Total Strings | Fitness = 46.85185185185185** |
| --- |
| 1. **{OM:9bds,FKt2Ue:46970104i}** 2. **{ +--:059/5::s,Jrr:6i}** 3. **{Yvt:9330i,\_E:K?x's}** 4. **{B7krMsk:12i,T0q:[@?`/;~s}** 5. **{+HMJXW4R:794s,U:864535i}** 6. **{tg48jr:95i,240\_0+4:6599s}** 7. **{u:3082s,I:80ORKXVqs}** 8. **{ag:{wfGfk7K:\*` ;s}}** 9. **{p+:h>Uzns,awTxHzXThz:9928s}** 10. **{Edgnz\_vke:'yHIUe@ruOs,2dSlu:i-ss}** 11. **{tt6cpxj93:665828153i,AGvbzdPxB: \\*<>[\_(]|s}** 12. **{oWx:FoKhNhKUs,QFU:wHCMDIs}** 13. **{YG:7s,W18A:-=.(^1's}** 14. **{kbczjCuk:cFUYs,Iq8e:005961916s}** 15. **{HXCHBL-U:tiqqhtqeQs,DpBGsMcmY:4900i}** 16. **{-:,<{t:{}, !OP !(I#l@W&J:{j:63i}}** 17. **{'U!9`h`g:{M+LT:4177015i}}** 18. **{U:6373s,nie.o:|gs}** 19. **{\_yM-GZu:9921i,4UP-:SfVzEs}** 20. **{ycagvgjsmy:zeiBotny)8s,A Y5:QvxcpXs}** 21. **{JZF:TzuTus,W1PHL6DY9Q:36i}** 22. **{d:57.F)3~;[os,JKJRSRPDS:709i}** 23. **{XF3HNBYP:! ]$ ^)`\_$s,\_:Zgc6GtgPqs}** 24. **{zcufxts15m:37;0s,J4Q:289i}** 25. **{TaKEFx:1s,++.+.- :878613s}** 26. **{20 SW8BR:39i,YDNcSSUXDM:(=\\*] s}** 27. **{fe\_:zas,vc58kgjos:02674811s}** 28. **{}** 29. **{lma:3\_`769+s,by:#/]\_!'\_|(s}** 30. **{ZdOLZnNd:1732814317i,zeekDaLRB:+?/<=#^s}** 31. **{NGH:+$~;?)<+s,BDSMSXPQNU:xHysKgxXKs}** 32. **{zl:nuDhVhrs,i9oQO:5778217i}** 33. **{q+f0x5z:15i,1zyagPVhul:548258i}** 34. **{33482214:6219703562i, .:U=FM>kP[Ns}** 35. **{SCXIIEV:/Yfs,rZSF:!9#>]~=s}** 36. **{HAC:IkgztEs,6E:ljghnBUsSs}** 37. **{sgiBuNWHa:~(1s,zRiEDYW:195368502s}** 38. **{..+ :5123i,98873:91i}** 39. **{}** 40. **{RvLJGWvG:IOs,Fjqwk:9\_;|3s}** 41. **{ykuyu:RRc6Zys, +.:^s}** 42. **{runfvzq:1028297324i,w:569423467i}** 43. **{uLAvOkt:X`)AtR)SJws,VAOLDCzo:k2r9qs}** 44. **{%""Z 8]]?B}Z.3:{}}** 45. **{h:62818780i,JIomM:-\*s}** 46. **{})=T#`:G:Q(O?#(]g,Y:{#(|>%\#L%..\*,$:{}}}** 47. **{FyNVGUM:!\_s,fgsejb:867134246i}** 48. **{D:75213785i,sfMN1j37:37944i}** 49. **{sepsxtdans:<i. C#R]s,XXHZ\_:379i}** 50. **{NSEK:.\_[<s,dzAIxASf:2&1>[<#s}** |

Figure 12: Best Performant String Set from Random Search

**Fitness Graphs**

**Hill Climb Graphs**

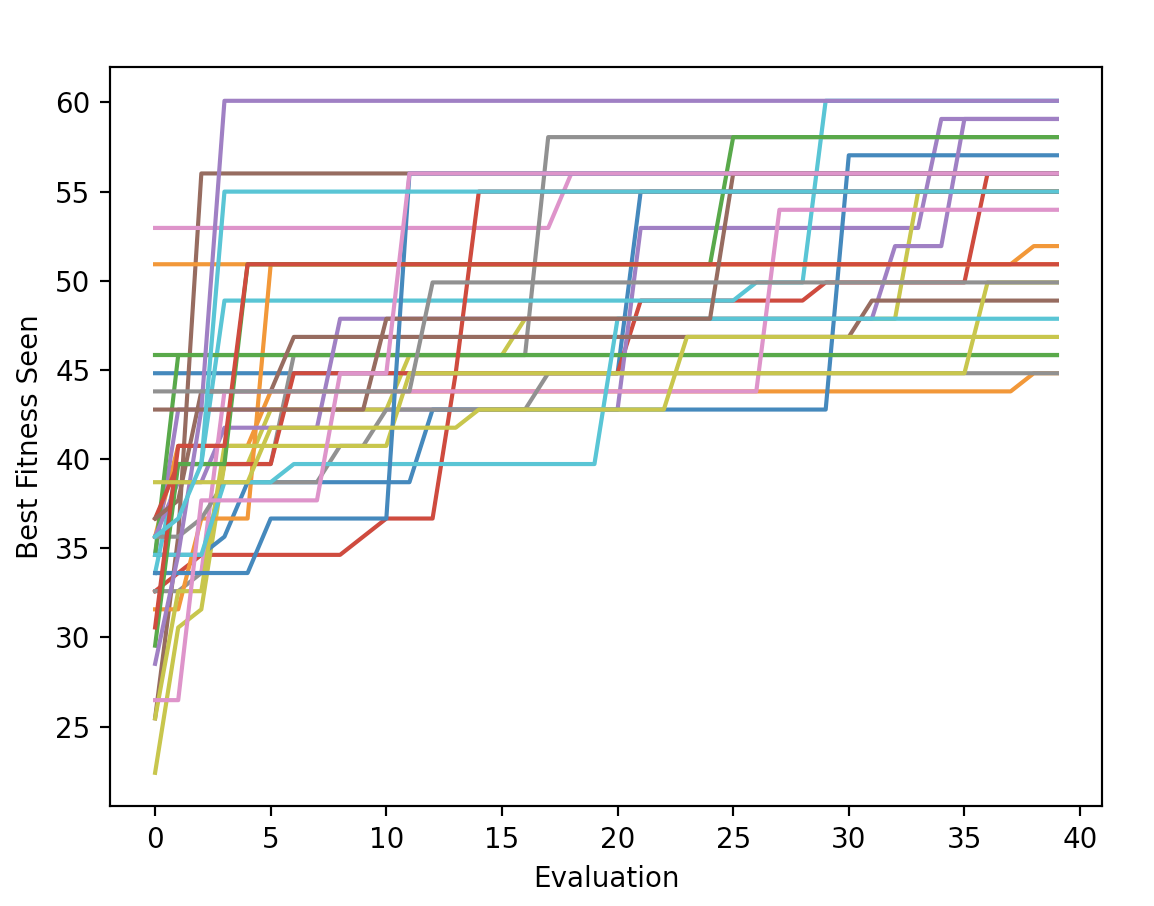
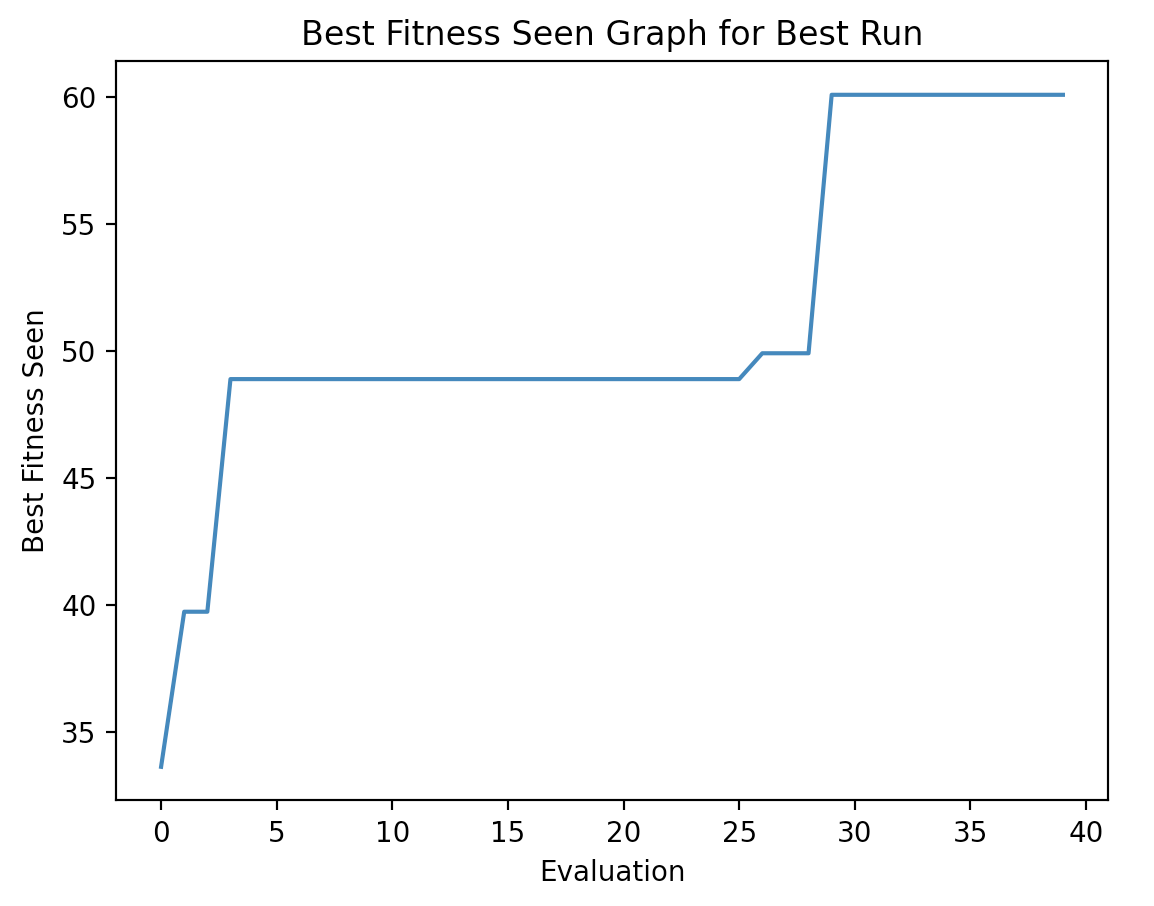
****

Figure 13: Hill ClimbBest Fitness of Best Run Figure 14: Hill ClimbBest Fitness for All Runs

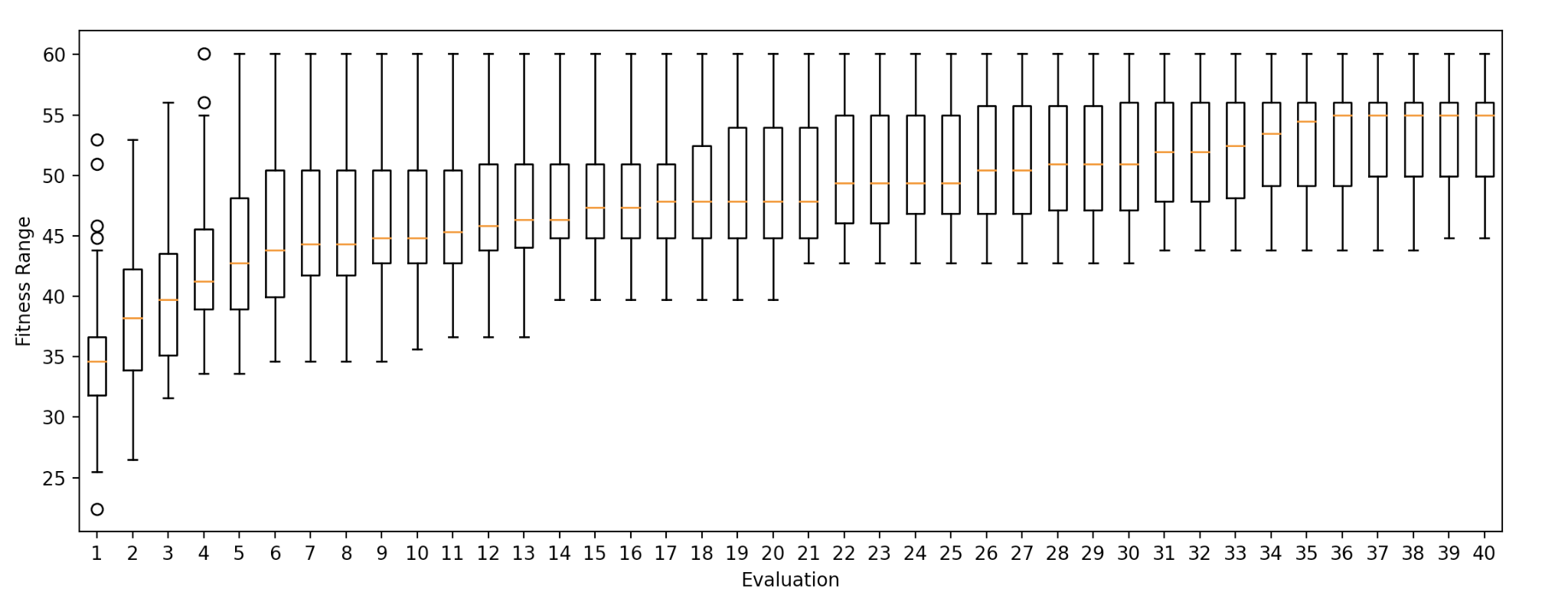
****

Figure 15: Hill ClimbVariation of Best Fitness across All Runs

**Random Search Graphs**

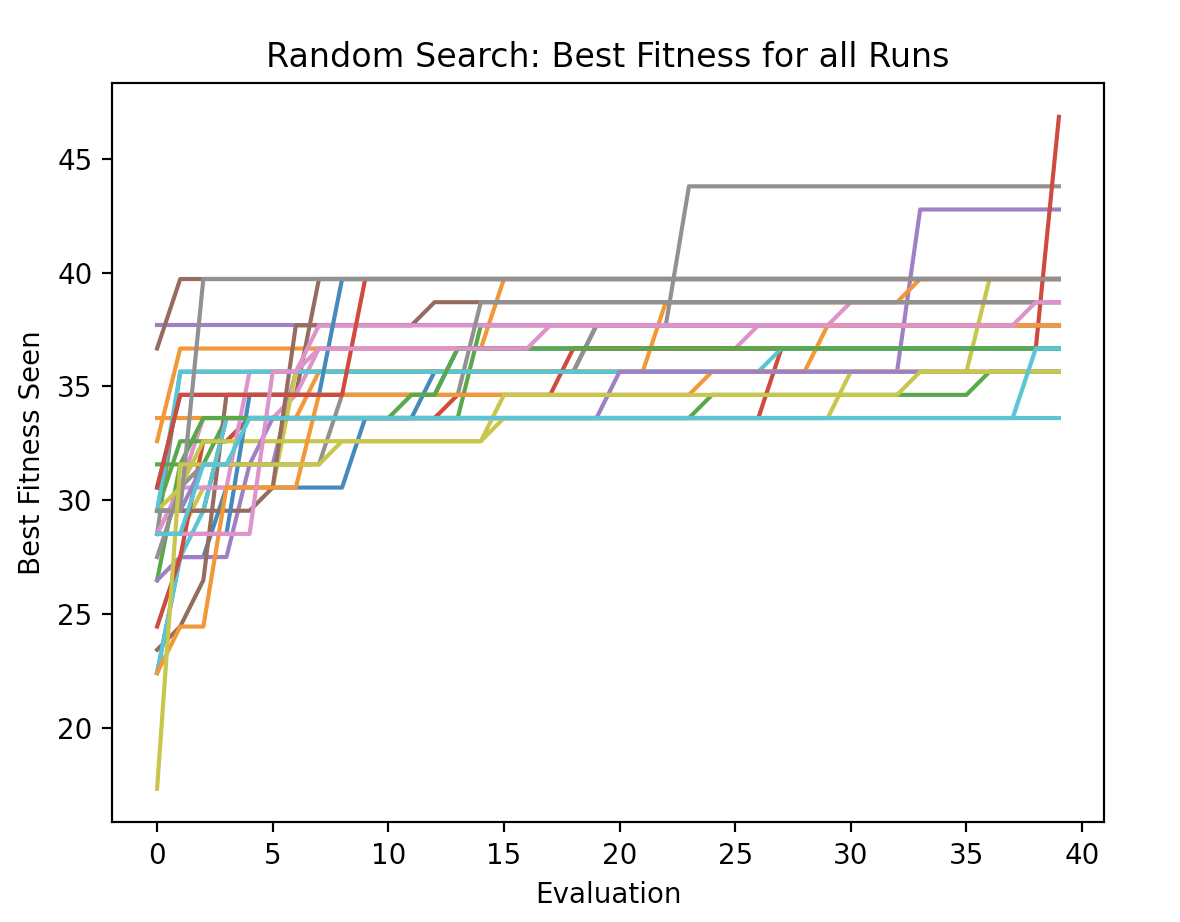
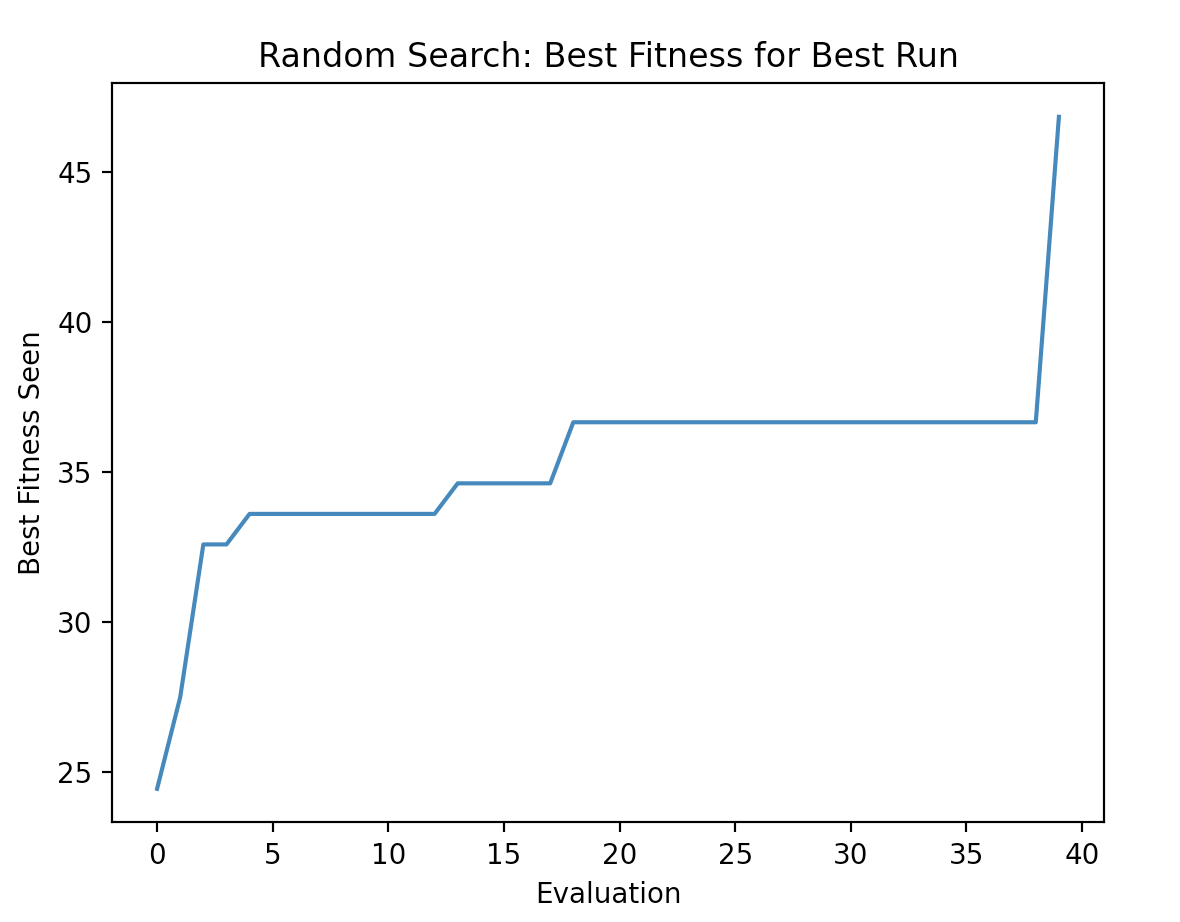
****

Figure 16: Random SearchBest Fitness of Best Run Figure 17: Random SearchBest Fitness for All Runs

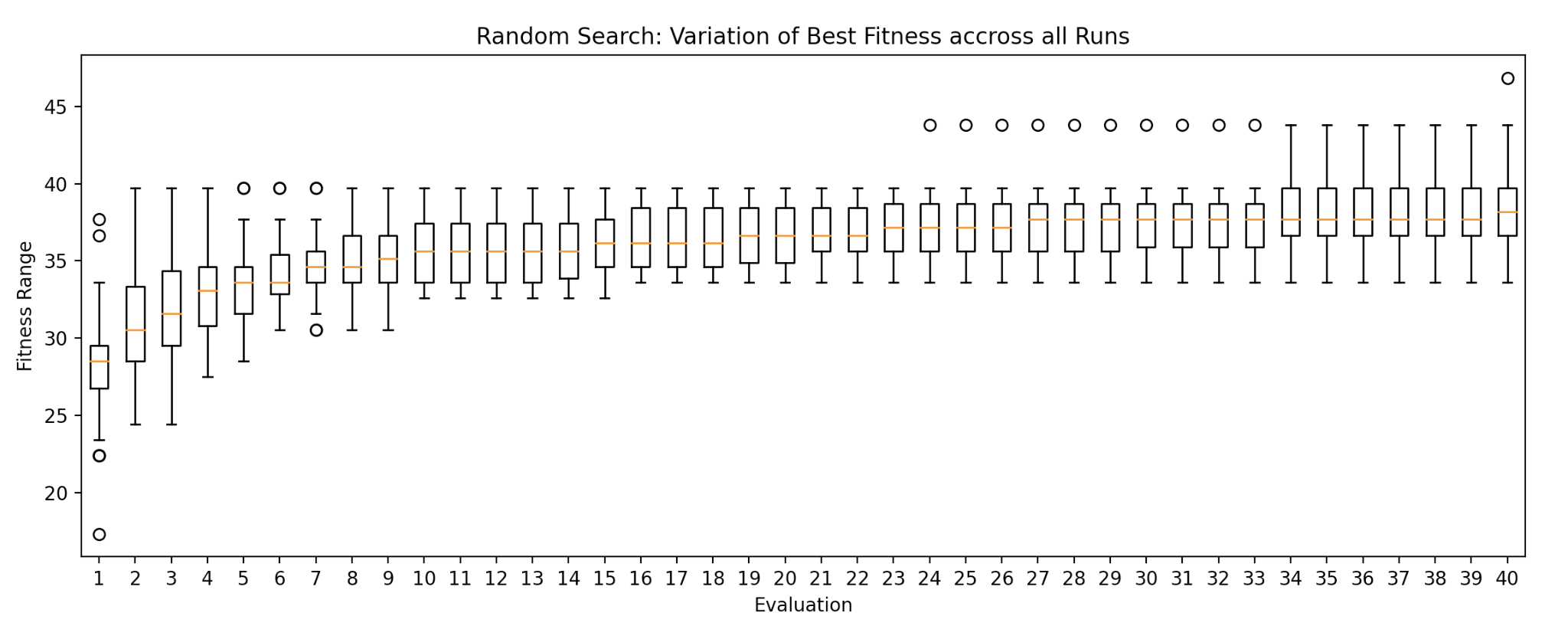
****

Figure 18: Random SearchVariation of Best Fitness across All Runs

## **Future improvements**

### Duplicate Substrings

Duplicate substrings were a trend we saw in some of the experiment’s best strings. While duplicate strings were accounted for, multiple strings with similar substrings might have the same effect of duplicate strings, but evaluating fitness in that respect would be heavily resource intensive and could not be implemented due to time constraints

### Sliding window / Preferred string recombination

One way of recombination we played around with was a sliding window approach of substrings with their individual evaluations and also having the fuzzing set contain a set of preferred strings in their genome in order to prefer better performing strings when recombination occurred. However individual string enumeration and the following fitness evaluation was much too resource intensive in our time window and could not be properly tested in time.

### Population Size

Increasing population size should help find even higher fitness values and lower the need for high mutation rates. This was unavailable to us for this paper as we lacked the computing power to do this in a time efficient manner without multithreading.

# 

# **Appendix A**

## **Random Search**

We start with Random search which generates and evaluates fuzzing with random generated strings.

For each implementation we send randomly generated strings (50 in our experiments) and record the exceptions thrown for each of the implementations. From the resulting matrix we calculate the fitness and plot the fitness graph.

## **Hill Climb**

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value.

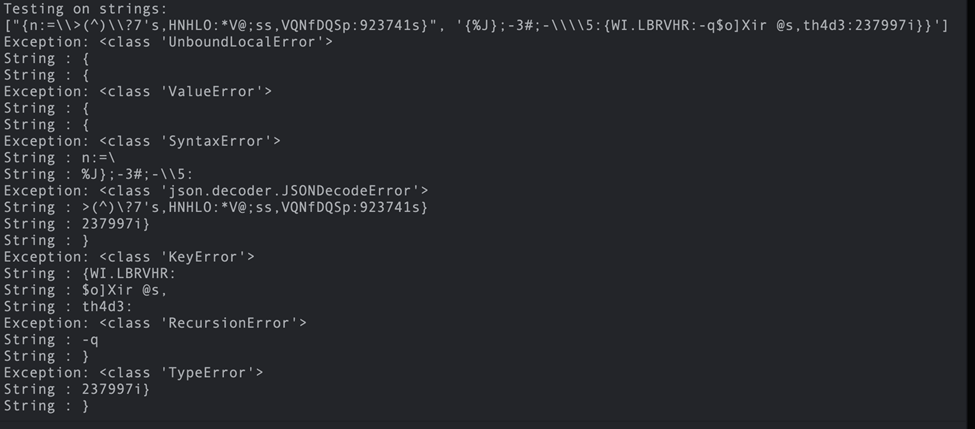
We used the Hill Climb search by first importing all the deserialization implementations, we then generate ‘n’ string set (50 strings in our experiment) and applied the set to each implementation. We record any exception thrown and calculate the fitness as described in the fitness evaluation section above.

# **Appendix B**

*1. Mutation and Neighbors: String Analysis*

In our string\_tester file, we iteratively rebuilt randomly generated high-fitness strings starting with an empty string and adding a character until reaching the original string, and tested its[DT1] exceptions thrown each time a new character was added, recording what parts of the string threw what exceptions. The overall takeaway from this analysis was that most if not all exceptions pertained to the nosj format, specifically variations in key-value pairs, inclusion of special characters in keys, and omission and slight variation of the format. This finding is what lead us to develop our neighbor and mutation functions accordingly.

*Example of string analysis function:*



*Neighbor Definition*

For our neighbor definition, we chose to define a neighbor of a given set as any set that is completely similar to the original set, except one key or value in a valid key-value pair is changed in any string. A valid key value pair is in the format { “\_\_” : “\_\_” }.

This format was deliberately chosen to not be specifically valid nosj, as with our previous random generation phase and further string analysis showed that completely valid nosj was prone to throwing only ‘Deserialization’ errors. Further, a looser formatting allows for a potentially wider coverage of string and mutation generation.