Mastermind jellegű karaktersor kitaláló genetikus algoritmus készítése Pythonban

Jakab Benedek (KZXLAC)

Célkitűzés

Egy olyan Python program készítése, ami képes paraméterezhető genetikus algoritmus segítségével kitalálni egy vagy több karaktersort.

Használati módok

A program meghatározó változója a useSingleString, amely eldönti, hogy egyetlen stringet generáljunk, amelynek hossza változtatható és azt találjuk meg genetikus algoritmussal, vagy több stringet, különbözően beállított algoritmusokkal, amelyről grafikont készítünk.

Paraméterek

stringLength: A generált string hossza

populationSize: A populáció mérete

iterations: Iterációk (generációk száma)

elitism: Elitizmus mértéke, az itt megadott számnyi fitness score alapján legjobb egyedből képzünk szülőket a crossoverekhez, illetve ezeket megtartjuk a populációban

crossoverNum: Ennyi crossovert képzünk egyszerre egy adott szülőpárosból (populáció és a generált elemek számának különbségének az osztójának kell lennie)

generation: Ennyi véletlenszerű lehetséges stringet képzünk a populáció végére, hogy az egyedek diverzitását fenntartsuk

mutationRate: Ekkora esélye van annak, hogy egy crossoveren belül egy adott karaktert mutáljunk

Segéd függvények

generateRandomString(length):

ASCII karakterek, számok és írásjelekből megadott paraméter hosszú véletlenszerű stringet képez.

fitnessFunction(candidate, target):

Kiszámítja egy adott string fitness scoreját azáltal, hogy összeszámolja hány karakter azonos a megfelelő pozícióban a paraméterül kapott jelölt (lehetséges megoldás) és a szintén paraméterül kapott cél (megoldás, amit szeretnénk megtalálni) között.

selectParents(population, fitnessScores, numParents):

A populációt először fitness scoreok alapján csökkenő sorrendbe rendezi, majd ennek az első numParents elemét visszaadja.

crossover(parent1, parent2):

A két szülőt egy véletlenszerű ponton elvágja, majd az elsőből származó rész mögé odailleszti a másodikból származó részt.

mutate(candidate, mutationRate):

Egy adott string karakterein végigiterál és az egyes karaktereit mutationRate eséllyel egy véletlenszerű karakterre változtatja.

A genetikus algoritmus

geneticAlgorithm(testString, populationSize, iterations, elitism, crossoverNum, generation, mutationRate):

Első lépésként képez egy kezdeti populációt véletlenszerű stringekkel. Utána iterációnként a következőket hajtja végre:

1. Kiszámolja az összes populációban lévő egyed fitness scoreját.
2. Kiválasztja az elitizmus mértéke szerint a lehetséges szülőket.

3. Képez populáció hossza - generált egyedek számú crossovert a következőképpen:
    a. Kiválaszt két véletlenszerű szülőt a potenciális szülőkből (akik eleve is benne maradnak a populációban)
    b. Majd ezen két szülőből crossoverNum crossovert képez, melyek mindegyikét mutálja is
4. A hátralévő üres helyekre új, véletlenszerűen generált stringeket helyez
5. Ellenőrzi, hogy a jelenlegi legjobb megoldás megegyezik-e a megoldással és, ha igen visszatér ennek értékével, nem fut feleslegesen és kiírja, hogy hányadik iterációban találta meg a legjobb megoldást, így könnyebb kiszűrni, ha feleslegesen sok iterációval próbálkoznánk.
6. Ha nem talált eddig (iterációk száma) teljesen megegyező megoldást, akkor visszaadja a jelenlegi legjobbat.

Több eset vizsgálata

Az eddigiek leírták, hogy egy stringet miként próbálunk kitalálni, nézzük azt az esetet, amikor több beállítást szeretnénk összehasonlítani.

Ehhez két extra paramétert vezettem be:

paramForGraph: meghatározhatjuk, hogy a több megoldást kiértékelő grafikon mely paraméter függvényében rajzolja ki a fitness függvény értékét.

parameterSets: Ezzel egy tömbön belül dictionaryk segítségével különböző beállításokat próbálhatunk ki, például: {"stringLength": 100, "populationSize": 100, "iterations": 100, "elitism": 20, "crossoverNum": 2, "generation": 10, "mutationRate": 0.01}

A grafikus megjelenítés, mivel kétdimenziós csak akkor használható megfelelően, ha ezen paraméterekből csak az egyiket változtatjuk egy adott futtatásnál.

Több beállításos esetben minden egyes beállításra lefuttatjuk az algoritmust, mindig más stringre, de ennek az eleve való véletlenszerűsége miatt nem tulajdonítok különösebb jelentőséget, illetve lehetséges azonos beállításokat különböző hosszú stringekre is tesztelni, így azok mindenképpen eltérnek. Az egyes megoldások legjobb

delikvensének fitness scoreját eltároljuk egy dictionarybe a grafikon másik paraméterének értékével együtt.

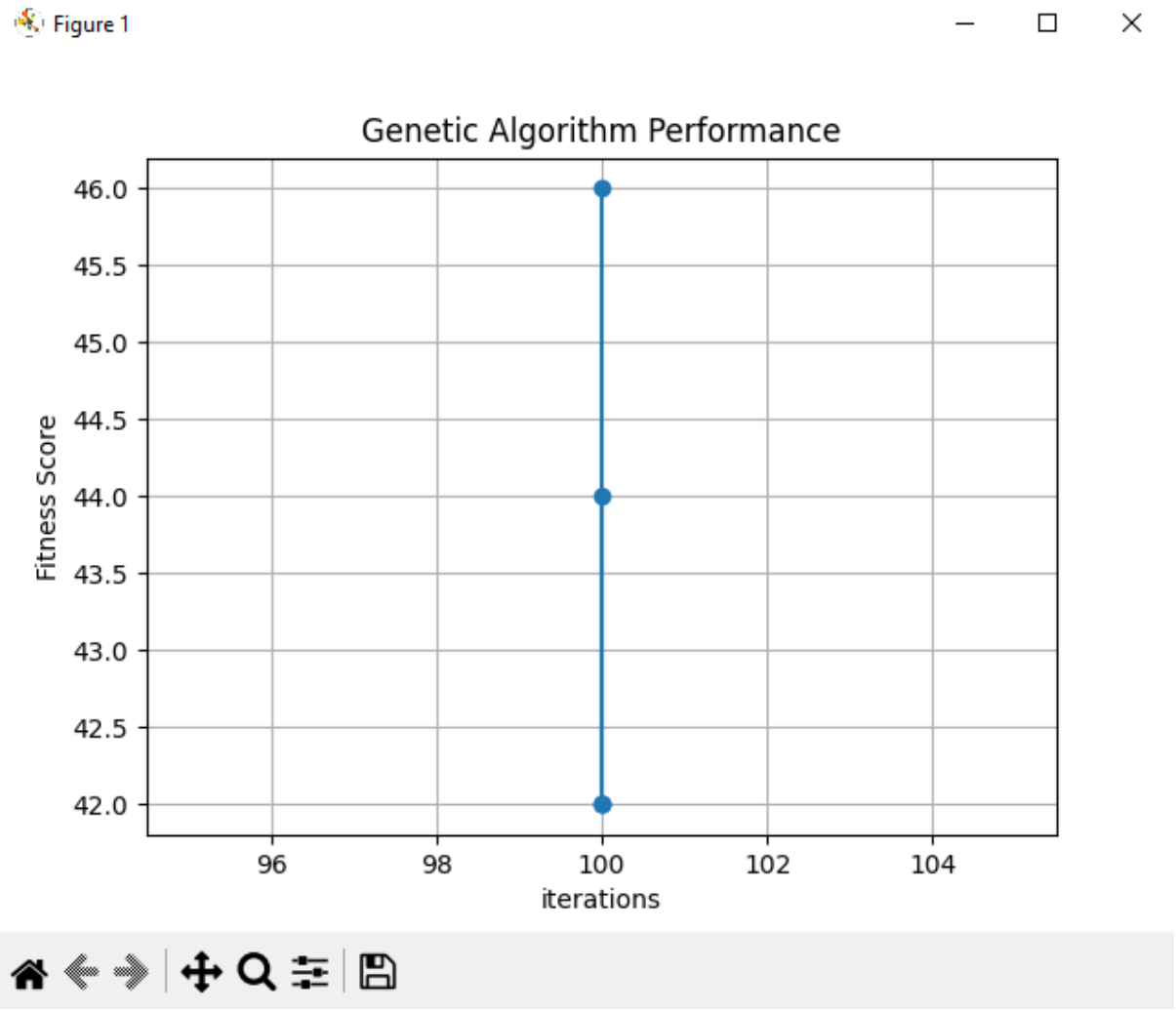
Ezt követően a generatePlot(results, paramForGraph) függvénnyel kirajzoljuk a megfelelő grafikont. Mivel azonos beállítások fitness scoreját más hosszúságú stringekkel nem éri meg összehasonlítani a fitness score és a string hosszának egyenes arányossága miatt, ezért ilyen esetben mindig a fitness score / string hossz értéket jeleníti meg, így látjuk, hogy mekkora részét találta el a generált stringnek.

Grafikonok

Mindig öt példával dolgozom, legyen az alapbeállítás a következő (ha explicit nem utalok másra, mindig ez az egyes vizsgálatok kiindulópontja):

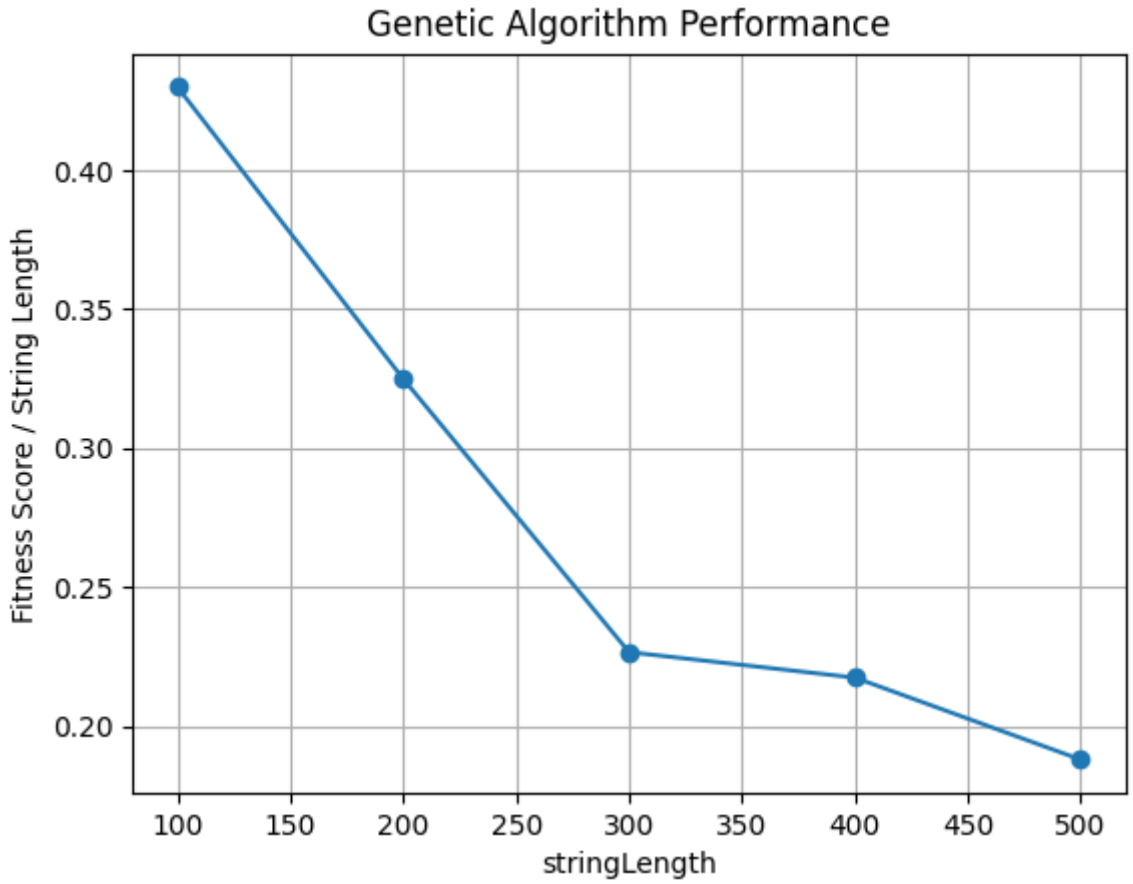
{"stringLength": 100, "populationSize": 100, "iterations": 100, "elitism": 20, "crossoverNum": 2, "generation": 10, "mutationRate": 0.01}

Többször futtatva a következő fitness score eredményeket kaptam:



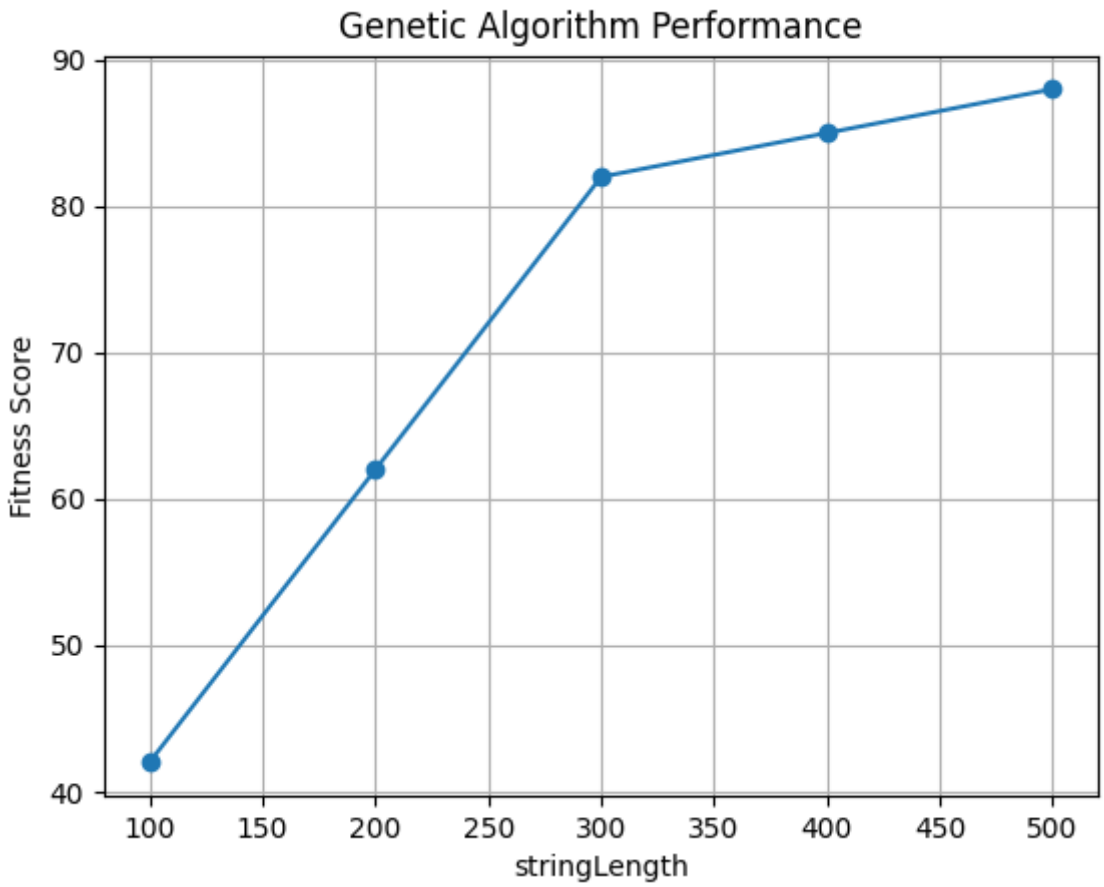
Ezen látszik, hogy ez a beállítás nagyjából 42-46 közötti fitness scoreokat eredményez (ezesetben lényegtelen, hogy mely paraméter függvényében, mivel mindegyik eset beállítása azonos).

Ellenőrizzük ezeket a beállításokat 100, 200, ..., 500 karakter hosszú stringekre.



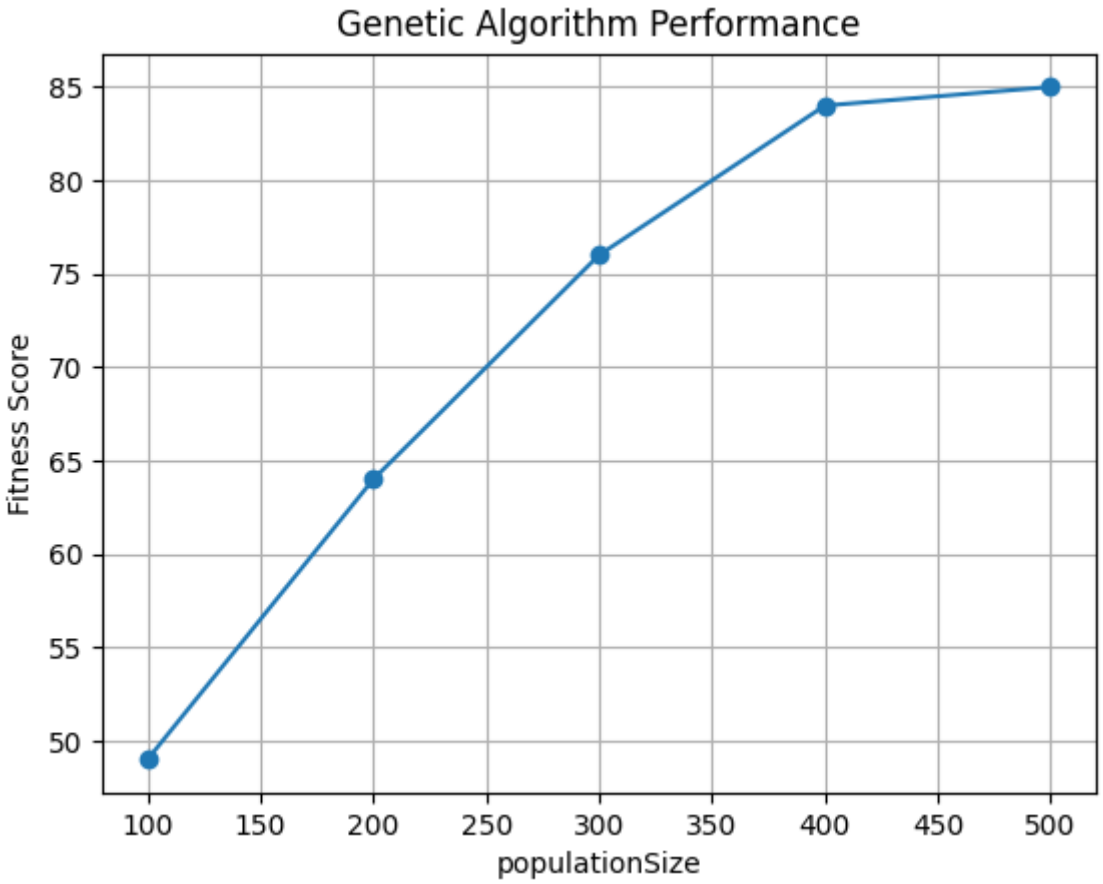
Látjuk, hogy azonos beállítások eltérő string hosszakra egyre kevésbé megfelelőek, amelyet főként a populáció nagysága és az iterációk száma befolyásol.

Érdekességképpen, ha a fitness scoret nem osztottam volna le a string hosszával:



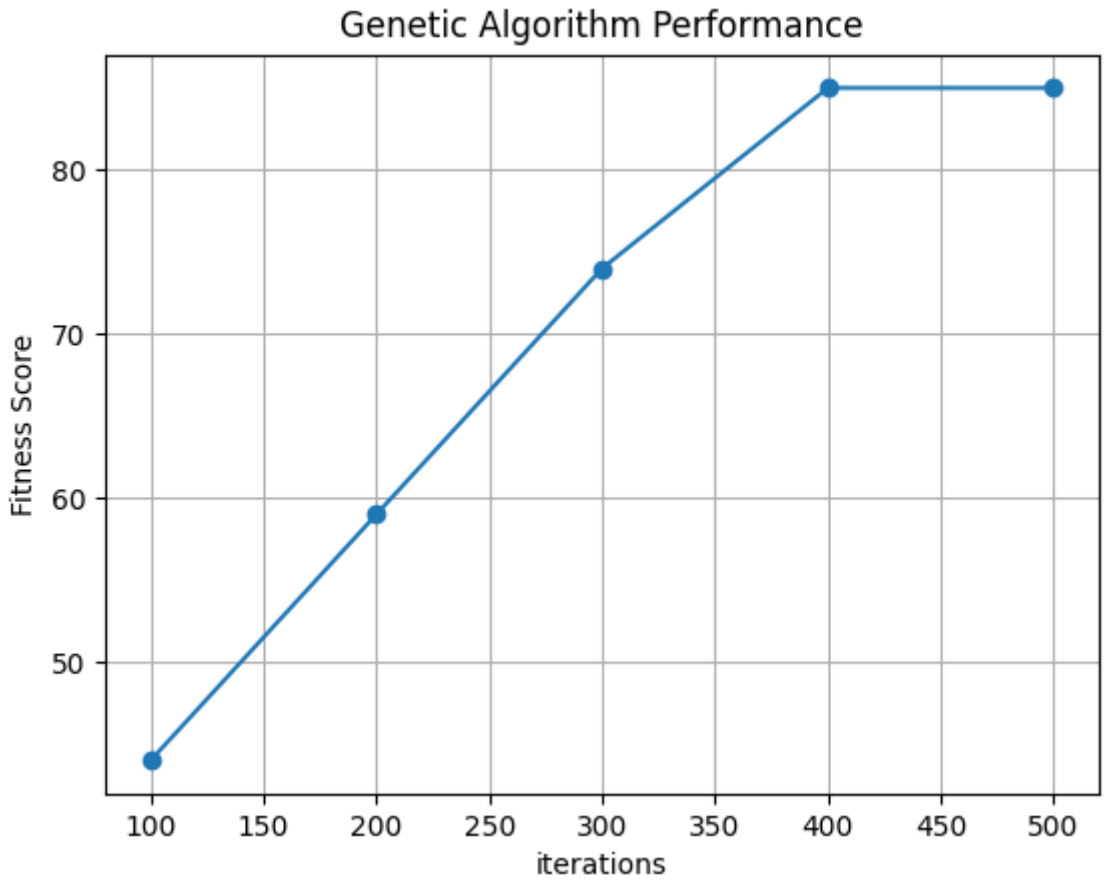
Látható, hogy a fitness scoreok emelkednek, de közel sem olyan ütemben, mint a string hossza.

Ellenőrizzük az eredeti a beállításokat 100, 200, ..., 500 méretű populációkra (vagyis a string hossza újra 100 minden esetben).



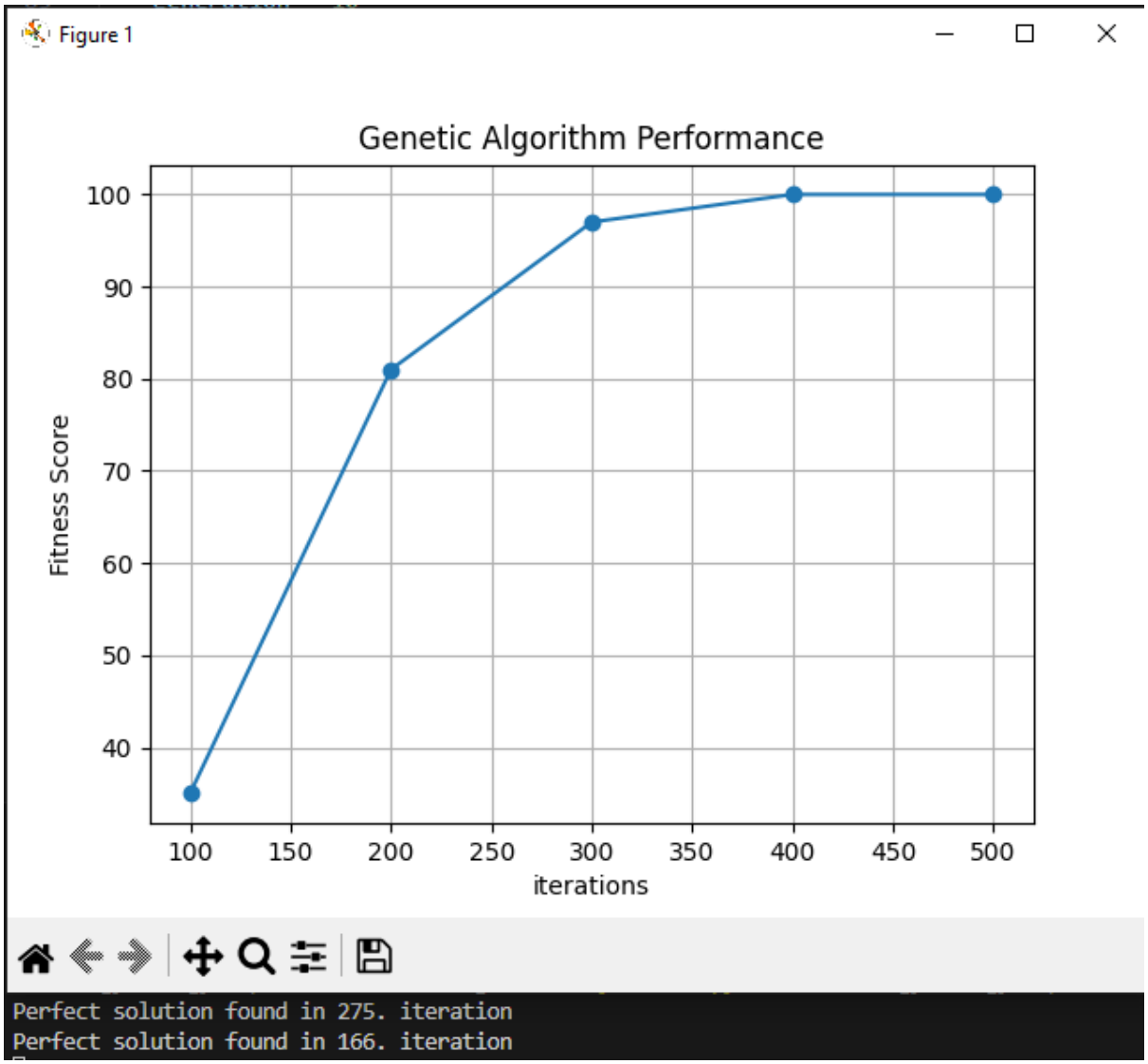
Gyökfüggvényre emlékeztető formában emelte az algoritmus teljesítményét, 500 iterációnál már majdnem eltalálta a megoldást.

Ellenőrizzük az eredeti a beállításokat 100, 200, ..., 500 iterációra.



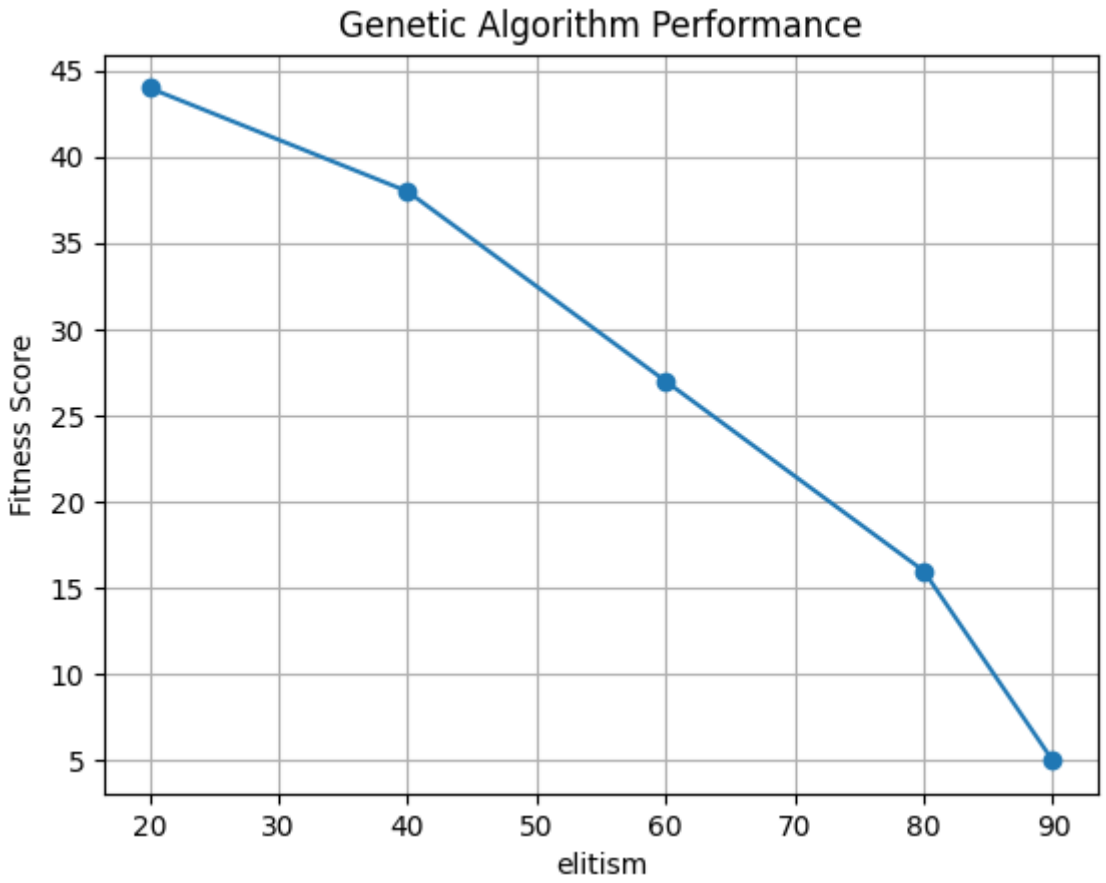
Szintén gyökfüggvényszerű emelkedés tapasztalunk.

Érdekességképpen az emelkedő iterációs beállításnál emelem a populáció nagyságát is azonos módon (100, 200, ..., 500).



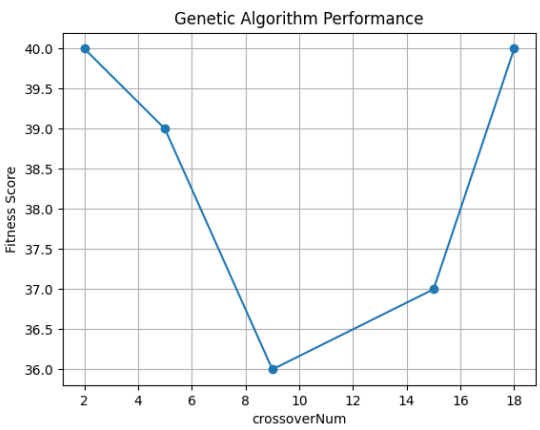
A populáció 400, illetve 500-ra emelése után már 275 és 166 iterációban alatt megtalálta a tökéletes megoldást, vagyis ebből arra következtetek, hogy a populáció nagyságának emelése és az iterációk emelése nem ajánlott azonos mértékben, mivel a populáció emelésével kevesebb iterációból megtalálható a tökéletes megoldás és ez felesleges iterációkat eredményezhet.

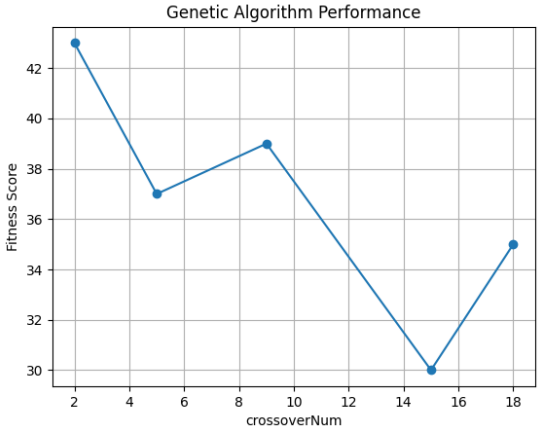
Visszaállítottam mindent az eredeti beállításokra. Ellenőrizzük az elitizmus hatását az algoritmus teljesítményére. 20, 40, 60, 80, 90 értékeket kapott. Az utolsó esetben azért nem 100, hogy a generált értékek általi diverzitást ne veszítsük el.

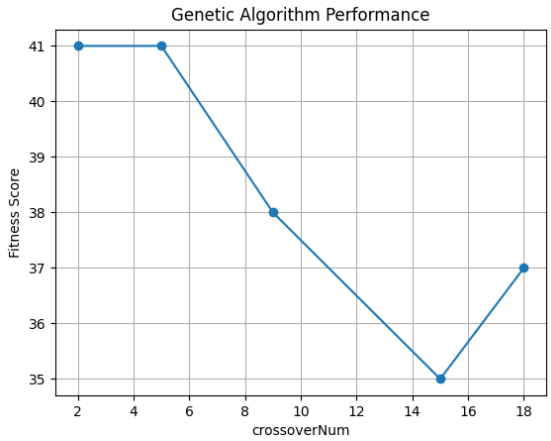


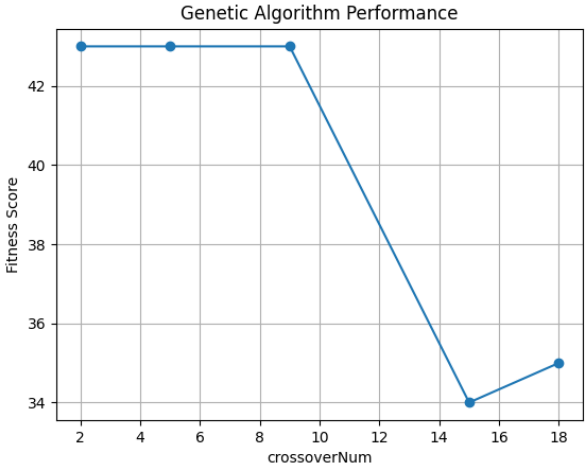
A növekvő elitizmus láthatóan rosszabb megoldást eredményezett, mivel minden esetben nagyon sok egyed maradt meg az első generálás után változatlanul.

Változtassuk a crossoverek számát. Mindig a populáció és a generált elemek számának különbségének az osztójának kell lennie. Esetünkben ez 100-10 = 90, miatt a 2, 5, 9, 15, 18-at választottam.



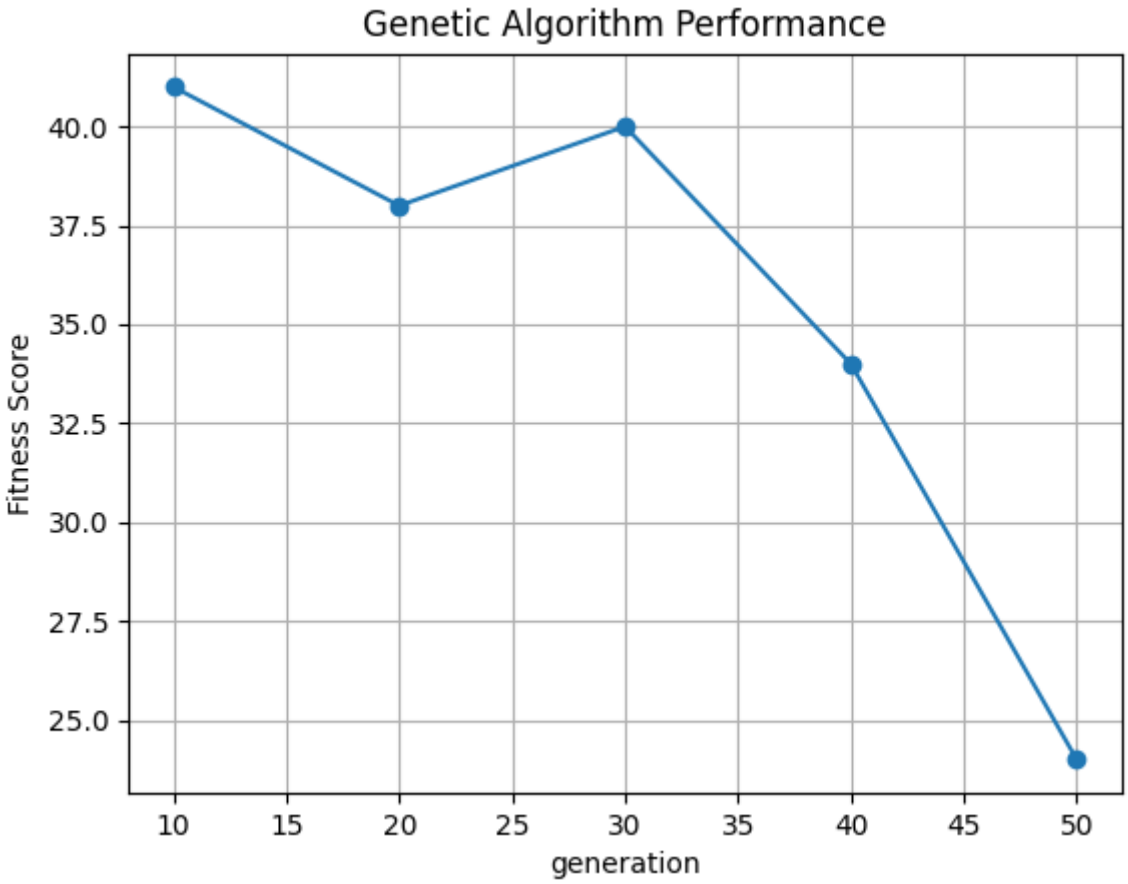






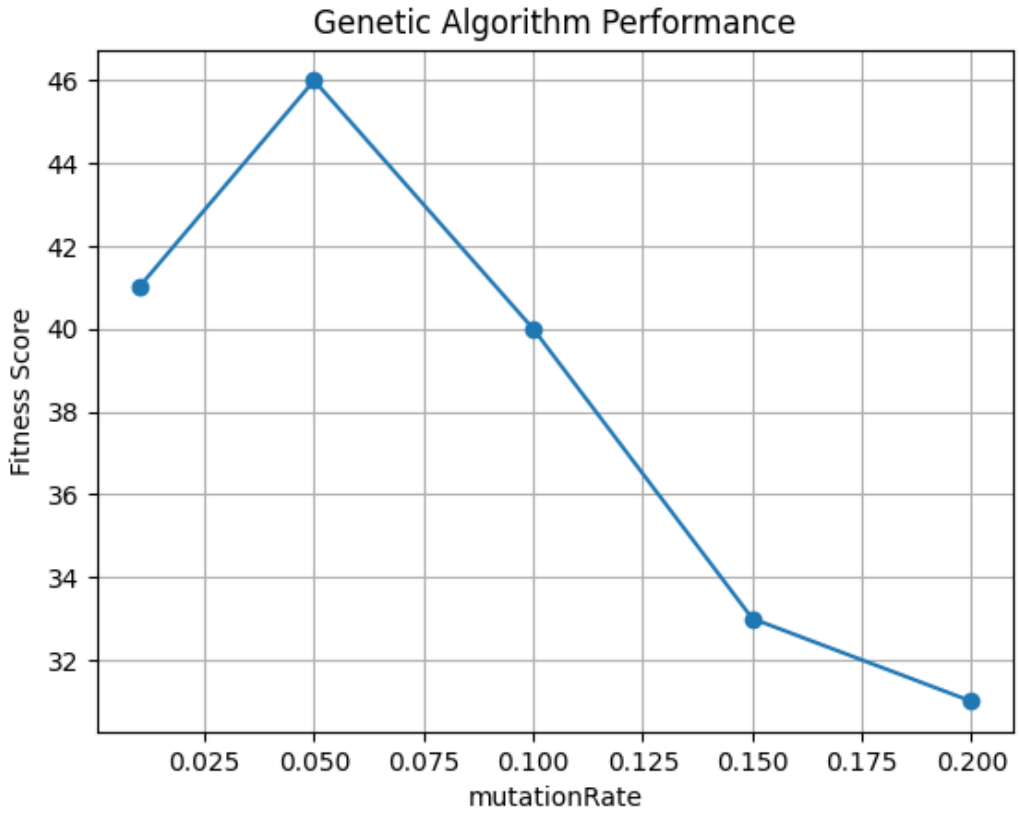
Az eredmények meglehetősen eltérőek, bár alapvetően csökkenő tendenciát mutatnak.

Változtassuk meg a generált értékek számát (10, 20, ..., 50).



Ez szintén csökkenő fitness scoret eredményezett, valószínűleg a túlzott mennyiségű véletlen és egymástól független opció miatt.

Végezetül a crossoverek mutációjának valószínűségét emeljük 0.01, 0.05, 0.10, 0.15, 0.20 értékekre. Többszöri próbálkozásból nyugodtan kijelenthetem, hogy az 5%-os mutációs esély adja a legjobb fitness score értékeket.



Ezek alapján egy legoptimálisabb beállításként a következőre következtetek egy 100 karakteres string kitalálására:

Population: 600, Iterations: 300, Mutation rate: 0.05, minden más az eredeti állapotban marad. Meglepő, de ezzel a beállítással maximum 95-ös fitness scoret tudtam elérni. A mutation rate 0.01-re való visszaállításával viszont potenciálisan 250 iterációból tökéletes megoldásra jutott az algoritmus. Ebből következően az általam megtalált egyik legoptimálisabb beállíáts egy 100 karakteres stringre:

stringLength = 100

populationSize = 600

iterations = 300

elitism = 20

crossoverNum = 2

generation = 10

mutationRate = 0.01