



Mesterséges Intelligencia Szorgalmi Feladat

Téma: Kör illesztése adott ponthalmazra 2.

Budapesti Műszaki és Gazdaságtudományi Egyetem

Gépészmérnöki Kar

Gyártástudomány és Technológia Tanszék

Készítette: Mile Kolos
Neptun-kód: OXEZ80
E-mail: kolosk5@gmail.com
Konzulens: Dr. Póka György

Budapest, 2025. november 18.

Tartalomjegyzék

1. A feladat leírása	2
1.1. A probléma bemutatása	2
1.2. Célkitűzések	2
2. A megoldás elve, módszere	3
2.1. Technológiai háttér	3
2.2. Ponthalmaz generálása hibákkal	3
2.3. A genetikus algoritmus mint megoldási módszer	3
3. Az implementáció bemutatása	5
3.1. A projekt struktúrája	5
3.2. <code>point_generator.py</code> : A ponthalmaz generátor	5
3.3. <code>genetic_algorithm.py</code> : A körillesztő algoritmus	6
3.4. <code>main.py</code> : A központi vezérlő szkript	7
3.5. <code>evaluation.py</code> : A kiértékelő modul	7
3.5.1. Ismert optimumú tesztek	8
3.5.2. Paraméter variációs tesztek	8
4. Tapasztalatok és kiértékelés	9
4.1. Tesztkörnyezet	9
4.2. Skálázhatóság és erőforrásigény	9
4.2.1. Futási idő	9
4.2.2. Memóriahasználat	10
4.3. Paraméterek érzékenységvizsgálata	11
4.4. Robusztusság és outlierok kezelése	11
4.5. Konvergencia vizsgálat	12
4.6. Validáció ismert optimumú teszteseteken	13
4.7. Futási idő és iterációszám kapcsolata	15
4.8. Paraméter variációk hatása konkrét példákra	15
4.9. Összegzés	16

1. A feladat leírása

1.1. A probléma bemutatása

A feladat egy klasszikus geometriai-optimalizálási probléma modern, mesterséges intelligencia alapú megközelítése. A cél egy adott 2D ponthalmazra a **legkisebb befoglaló kör** (minimum enclosing circle) meghatározása. A kihívást az jelenti, hogy a ponthalmaz nem ideális, hanem valós mérési vagy digitalizálási folyamatokból származó hibákat szimulál, úgymint:

- **Alakhiba:** A pontok nem egy tökéletes köríven helyezkednek el.
- **Véletlen zaj:** Minden pont pozíciója egy kis mértékű, véletlenszerű eltolással terhelt.
- **Kiugró pontok (outlierek):** A ponthalmaz tartalmaz néhány, a fő csoporttól távol eső, hibás mérési eredményt szimuláló pontot.

Az algoritmusnak robusztusnak kell lennie, hogy ezekkel a hibákkal megbirkózzon, és a definíció szerint megtalálja azt a legkisebb sugarú kört, amely a ponthalmaz *összes* elemét tartalmazza.

1.2. Célkitűzések

A projekt során a következő fő célokat kellett elérni:

1. **Ponthalmaz generálása:** Egy olyan programmodul létrehozása, amely képes paramétereizhető módon, a fent említett hibákkal terhelt ponthalmazokat generálni.
2. **Algoritmus fejlesztése:** Egy mesterséges intelligencia alapú algoritmus (esetünkben genetikus algoritmus) implementálása, amely a generált ponthalmazra illeszti a legkisebb befoglaló kört.
3. **Kiértékelés:** A kifejlesztett módszer teljesítményének objektív elemzése. Vizsgálni kell a futási időt a bemeneti adatok méretének függvényében, elemezni kell az algoritmus paramétereinek (pl. mutációs ráta) hatását az eredményre, és vizsgálni kell a megoldás konvergenciáját.
4. **Dokumentálás:** A teljes folyamat, a módszer és az elért eredmények részletes dokumentálása a követelményeknek megfelelően.

2. A megoldás elve, módszere

A probléma megoldására egy Python-alapú szoftveres megoldás készült, amely egy genetikus algoritmust alkalmaz a legkisebb befoglaló kör megkeresésére.

2.1. Technológiai háttér

A projekt az alábbi technológiákra épül:

- **Nyelv:** Python 3
- **Könyvtárak:**
 - **NumPy:** A numerikus számítások (távolságmérés, koordináta-manipuláció) hatékony elvégzéséért felel. Nélkülözhetetlen a nagy mennyiségű pontadat gyors feldolgozásához.
 - **Matplotlib:** Az adatok és eredmények vizualizációjáért felel. Segítségével ábrázoljuk a generált ponthalmazokat, az illesztett köröket és a kiértékelés során kapott grafikonokat.

2.2. Ponthalmaz generálása hibákkal

A kiindulási adathalmazt egy dedikált modul (`point_generator.py`) hozza létre, amely egy ideális körből indul ki, és szisztematikusan hibákat ad hozzá:

1. **Alap kör definiálása:** Egy (x, y) középpontú, r sugarú körön egyenletesen elhelyezzünk N számú pontot.
2. **Alakhiba hozzáadása:** A pontok koordinátáit enyhén torzítjuk, ami egy ellipszisszerű alakot eredményez.
3. **Véletlen zaj hozzáadása:** Minden pont x és y koordinátájához egy normális eloszlású véletlen értéket adunk, ami a mérési pontatlanságot szimulálja.
4. **Kiugró pontok generálása:** A fő ponthalmazon kívül, nagyobb távolságra elhelyezzünk néhány pontot, amelyek a durva mérési hibákat reprezentálják.

2.3. A genetikus algoritmus mint megoldási módszer

A legkisebb befoglaló kör egy optimalizálási problémaként fogható fel, amelynek megoldására a genetikus algoritmus (GA) kiválóan alkalmas. A GA az evolúció elveit (szelekció, keresztezés, mutáció) utánózva keresi a legjobb megoldást.

- **Egyed (Individuum):** Egy lehetséges megoldás, esetünkben egy kör. Minden egyedet három gén ír le: a kör középpontjának cx és cy koordinátája, valamint r sugara. A kromoszóma tehát egy $[cx, cy, r]$ számtömb.
- **Populáció:** Egyedek (körök) egy csoportja, amelyekkel az algoritmus egy adott generációban dolgozik.
- **Fitnessfüggvény:** Az algoritmus legkritikusabb része. Megmondja, hogy egy adott kör (egyed) mennyire "jó" megoldás. A cél a sugár minimalizálása, miközben az összes pont a körön belül helyezkedik el. A fitness értékét a következőképpen számoljuk:

$$\text{fitness} = \text{sugár} + \text{büntetés}$$

- A **büntetés** értéke nulla, ha az összes pont a körön belül van.
- Ha egy vagy több pont a körön kívülre esik, a büntetés arányos a körvonalon kívül eső pontok távolságainak összegével. Ez a büntetőtag "kényszeríti" az algoritmust, hogy olyan köröket részesítsen előnyben, amelyek minden pontot lefednek.
- **Evolúciós folyamat:**
 1. **Inicializálás:** Véletlenszerű körökből álló kezdő populáció létrehozása.
 2. **Szelekció:** A populációból a legjobb fitnessértékkel (legkisebb értékkel) rendelkező egyedek kiválasztása. A jobb egyedek nagyobb eséllyel vesznek részt a szaporodásban.
 3. **Keresztezés (Crossover):** Két kiválasztott szülőegyed génjeinek (koordináták, sugár) kombinálásával új utódok jönnek létre.
 4. **Mutáció:** Az utódok génjeit kis, véletlenszerű mértékben módosítjuk. Ez biztosítja a változatosságot és segít elkerülni a lokális optimumokba való beragadást.
 5. **Ismétlés:** A folyamat a szelekciótól a mutációig ismétlődik egy előre meghatározott generációszámon keresztül.
- **Leállási feltétel:** Az algoritmus a megadott számú generáció lefutása után leáll, és a legjobb addig talált egyedet (kört) adja vissza megoldásként.

3. Az implementáció bemutatása

A szoftver Python 3 nyelven készült, moduláris felépítéssel, hogy a különböző funkciók (pontgenerálás, algoritmus, kiértékelés) logikailag elkülönüljenek.

3.1. A projekt struktúrája

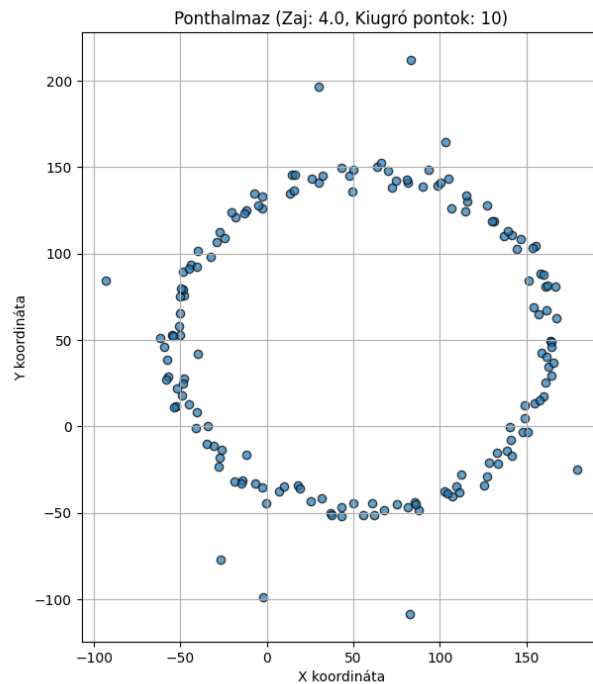
A projekt fő mappái és fájllai a következők:

```
.
|-- docs/                # Dokumentáció és feladatkiírás
|-- src/                 # A Python forráskódok
|   |-- point_generator.py
|   |-- genetic_algorithm.py
|   |-- evaluation.py
|   |-- main.py
|-- venv/                # Virtuális környezet
|-- .gitignore
|-- requirements.txt     # Projekt függőségek
|-- README.md
```

3.2. point_generator.py: A ponthalmaz generátor

Ez a modul felel a tesztadatok, azaz a hibákkal terhelt ponthalmazok létrehozásáért.

- **generate_point_cloud(...)**: A központi függvény, amely a megadott paraméterek (középpont, sugár, pontszám, hibák mértéke) alapján legenerálja és visszaadja a pontokat egy NumPy tömbben.
- **visualize_point_cloud(...)**: Egy segédfüggvény, amely a Matplotlib segítségével kirajzolja a generált pontokat.



1. ábra. A generált pontthalmaz vizualizációja

3.3. genetic_algorithm.py: A körillesztő algoritmus

Ez a fájl tartalmazza a genetikus algoritmus logikáját egy **CircleGA** nevű osztályba zárva.

- **CircleGA.__init__(...)**: A konstruktor inicializálja az algoritmust a megadott paraméterekkel (populációméret, mutációs ráta stb.), és létrehozza a kezdeti, véletlenszerű körökből álló populációt.
- **CircleGA._calculate_fitness()**: Az algoritmus legfontosabb metódusa. Kiszámítja minden egyes körhöz a fitness értéket. A fitness a kör sugarából és egy büntetőtagból áll.

```
# Részlet a fitness számításból
distances = np.sqrt((self.points[:, 0] - cx)**2 + (self.
    points[:, 1] - cy)**2)

# Büntetés a körön kívül eső pontokért
outside_points = distances[distances > r]

# A büntetés súlyozása
penalty = np.sum(outside_points - r) * 10

# A fitness a sugár és a büntetés összege
fitness_scores[i] = r + penalty
```

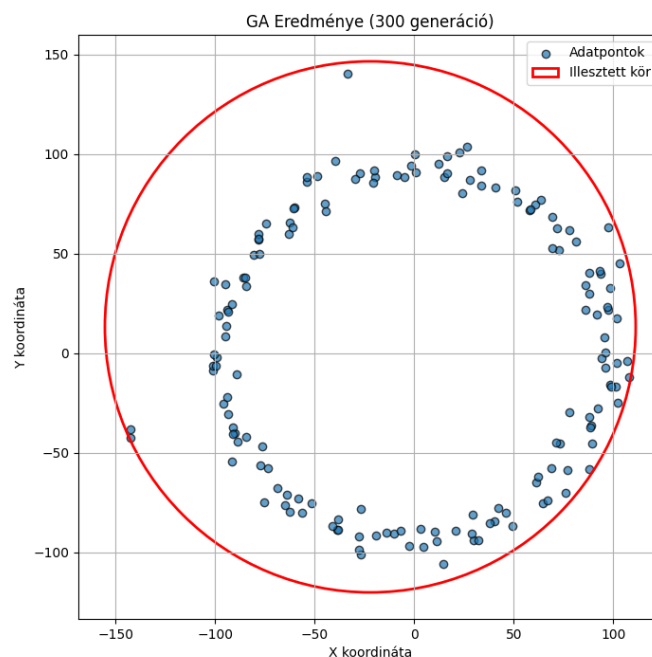
- **CircleGA._select(), _crossover(), _mutate()**: Ezek a metódusok valósítják meg a klasszikus evolúciós lépéseket.

- **CircleGA.run()**: A fő ciklus, amely a generációkon keresztül futtatja az evolúciót, és a végén visszaadja a legjobb megtalált kört és a konvergencia-történetet.

3.4. main.py: A központi vezérlő szkript

Ez a szkript a program belépési pontja. Összefogja a többi modul működését, és lehetővé teszi a program parancssori futtatását és paraméterezését.

- **Argumentumok feldolgozása:** Az argparse könyvtár segítségével kezeli a parancssori argumentumokat, így a pontgenerálás és a genetikus algoritmus minden fontos paramétere könnyen módosítható futás közben.
- **Vezérlés:** Meghívja a pontgenerátort, majd az eredményül kapott pontokra ráfuttatja a genetikus algoritmust.
- **Eredményközlés:** Kiírja a konzolra a futási időt és a megtalált kör paramétereit, majd vizuálisan is megjeleníti az eredményt.



2. ábra. A program futásának eredménye: az illesztett kör

3.5. evaluation.py: A kiértékelő modul

Ez a modul az algoritmus teljesítményének részletes, statisztikai alapú elemzésére szolgál. A modul képes automatikusan lefuttatni a különböző tesztek, az eredményeket CSV fájlba menteni, és professzionális grafikonokat generálni a dokumentációhoz.

A modul parancssori argumentumokkal vezérelhető:

- **-all:** Az összes teszt futtatása.

- **-scalability:** Skálázhatósági teszt (futási idő és memória a pontszám függvényében).
- **-mutation:** Mutációs ráta érzékenységvizsgálata.
- **-robustness:** Robusztusság vizsgálata (outlierek hatása).
- **-convergence:** Konvergencia vizsgálat.
- **-known-optimum:** Ismert optimumú tesztesetek validálása.
- **-iteration-runtime:** Futási idő és iterációszám kapcsolata.
- **-parameter-variation:** Különböző paraméter-beállítások vizuális bemutatása.

3.5.1. Ismert optimumú tesztek

A modul tartalmaz speciális geometriai alakzatokat generáló függvényeket, amelyeknél az optimális befoglaló kör analitikusan ismert:

- **Szabályos sokszögek:** Háromszög, négyzet, hatszög, tízszög – a körülírt kör sugara egyenlő a csúcsok középponttól mért távolságával.
- **Ellipszisek:** Az optimális befoglaló kör sugara a nagyobb féltengely hosszával egyenlő.

Ezek a tesztek lehetővé teszik az algoritmus pontosságának objektív mérését.

3.5.2. Paraméter variációs tesztek

A modul különböző nehézségű pontthalmazokat generál a zaj, alakhiba és outlierek kombinálásával, majd vizuálisan összehasonlítja az eredményeket. Ez szemléletesen bemutatja az algoritmus robusztusságát.

A modul a `tracemalloc` könyvtárat használja a memóriahasználat mérésére, és a `pandas` segítségével kezeli a mérési adatokat. Az eredmények a `docs/documentation/data` (CSV) és `docs/documentation/images` (PNG) mappákba kerülnek.

4. Tapasztalatok és kiértékelés

A szoftver fejlesztésének utolsó fázisában egy átfogó teljesítmény- és minőségvizsgálatot végeztünk. A kiértékelés célja annak igazolása volt, hogy a genetikus algoritmus képes megbízhatóan megtalálni a legkisebb befoglaló kört különböző nehézségű bemeneti adatok esetén is, valamint a futási idő és memóriaigény alakulásának feltérképezése.

Mivel a genetikus algoritmusok sztochasztikus (véletlen alapú) működésűek, egyetlen futtatás eredménye nem tekinthető mérvadónak. Ezért a kiértékelés során statisztikai megközelítést alkalmaztunk: minden mérést többször (10-30 alkalommal) megismételtünk, és az eredmények átlagát, valamint szórását vizsgáltuk.

4.1. Tesztkörnyezet

A méréseket az alábbi hardver- és szoftverkörnyezetben végeztük:

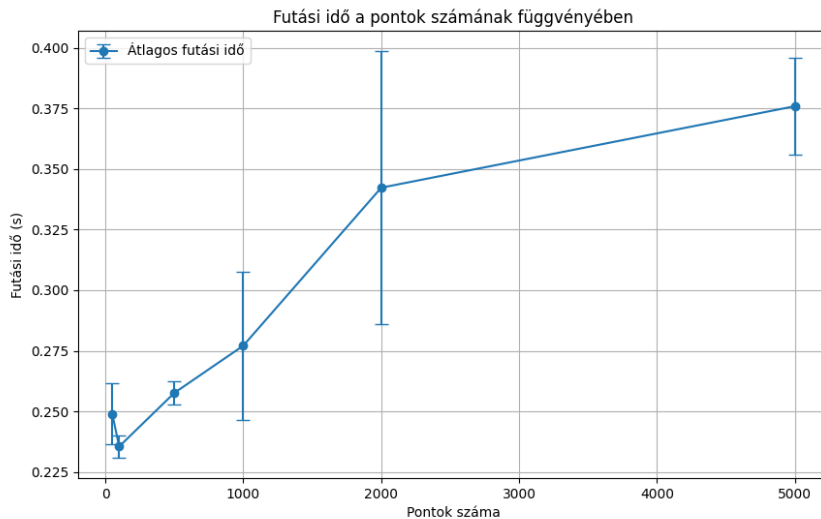
- **Hardver:** 12th Gen Intel(R) Core(TM) i7-12700H CPU, 32 GB RAM
- **Operációs rendszer:** Windows 11
- **Környezet:** Python 3.12
- **Felhasznált könyvtárak:** NumPy (számítások), Matplotlib (vizualizáció), tracemalloc (memóriamérés).

4.2. Skálázhatóság és erőforrásigény

Az első vizsgálat célja annak meghatározása volt, hogyan növekszik a futási idő és a memóriahasználat a bemeneti pontok számának (N) növelésével. A teszt során 50 és 5000 közötti pontszámokat vizsgáltunk.

4.2.1. Futási idő

A futási idő mérése a teljes algoritmus lefutását magában foglalta (populáció inicializálása + generációk futtatása).

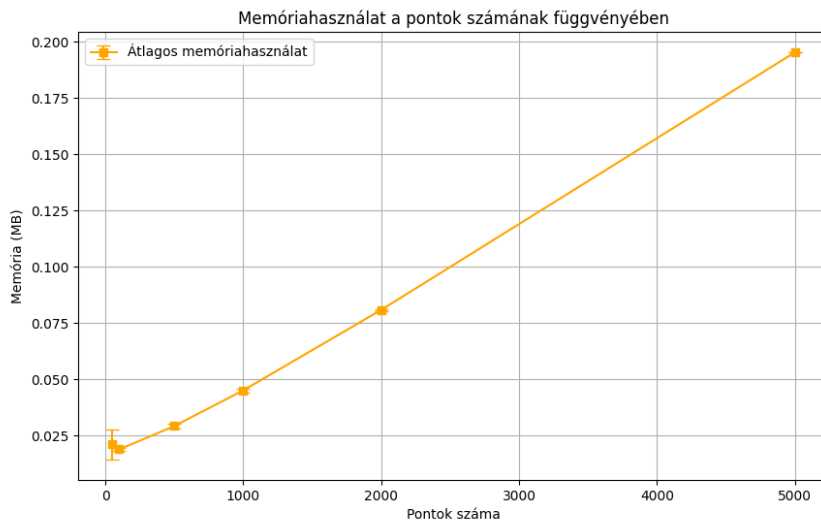


3. ábra. Futási idő a pontok számának függvényében

Elemzés: A grafikonon látható, hogy a futási idő a pontok számával közel lineárisan, vagy enyhén polinomiálisan növekszik. Ez kedvező eredmény, mivel azt mutatja, hogy az algoritmus nagyobb adathalmazok (pl. 5000 pont) esetén is kezelhető időn belül (néhány másodperc alatt) eredményt ad. A szórás (hibasávok) viszonylag kicsi, ami stabil futási teljesítményre utal.

4.2.2. Memóriahasználat

A memóriahasználatot a Python `tracemalloc` moduljával mértük, amely a program futása során lefoglalt maximális memóriát (peak memory) rögzítette.



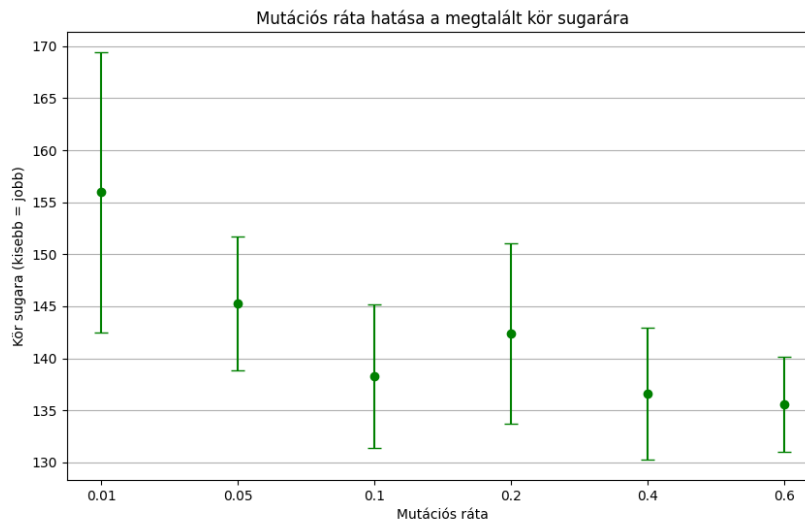
4. ábra. Memóriahasználat a pontok számának függvényében

Elemzés: A memóriahasználat szintén növekvő tendenciát mutat, de abszolút értékben rendkívül alacsony marad (még 5000 pont esetén is 1 MB alatt/körül mozog a többlet

memóriaigény). Ez azt bizonyítja, hogy az implementáció memóriahatékony, a NumPy tömbök használata optimalizált adattárolást tesz lehetővé.

4.3. Paraméterek érzékenységvizsgálata

A genetikus algoritmusok teljesítményét nagyban befolyásolják a hiperparaméterek. Kiemelten vizsgáltuk a **mutációs ráta** hatását, mivel ez felelős a populáció sokszínűségének fenntartásáért és a lokális optimumok elkerüléséért.



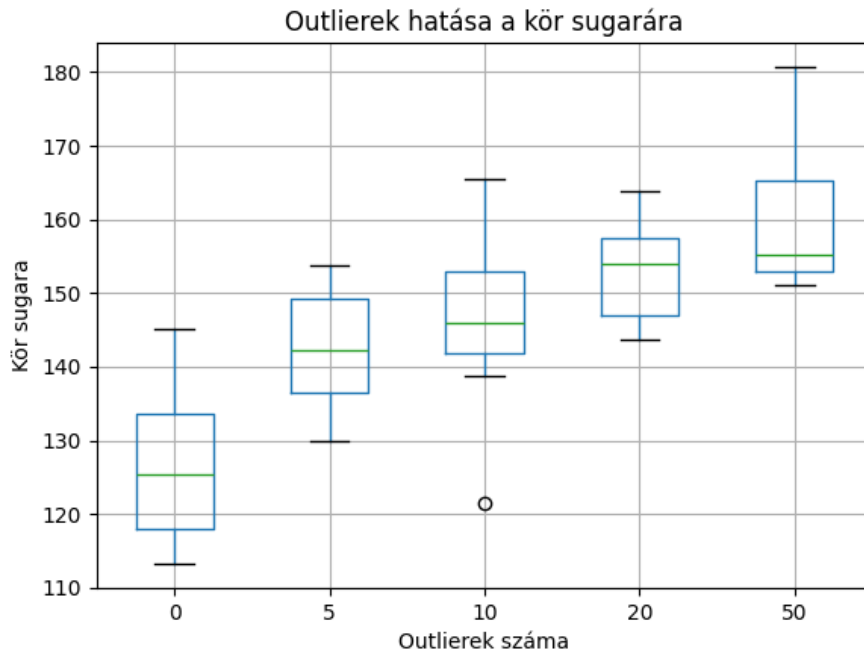
5. ábra. Mutációs ráta hatása

Elemzés: A mérési eredmények (zöld pontok és hibasávok) alapján a következőket állapíthatjuk meg:

- **Túl alacsony mutáció (0.01):** Az algoritmus hajlamos korán konvergálni egy lokális optimumba, így a megtalált kör sugara nagyobb (rosszabb), mint az optimális.
- **Optimális tartomány (0.1 - 0.2):** Ebben a tartományban a legkisebb a megtalált körök sugara. Itt az algoritmus egyensúlyt tart a felfedezés (exploration) és a kiaknázás (exploitation) között.
- **Túl magas mutáció (> 0.4):** A keresés véletlenszerűvé válik, a jó megoldások "szétesnek" a túlzott változtatások miatt, így az eredmények romlanak és a szórás is megnő.

4.4. Robusztusság és outlierok kezelése

A feladatkiírás egyik kritikus pontja a hibás mérések (outlierek) kezelése volt. Mivel a legkisebb befoglaló kör definíció szerint *minden* pontot tartalmaz, egyetlen távoli pont is drasztikusan megnövelheti a szükséges kör sugarát. Azt vizsgáltuk, hogy az algoritmus képes-e alkalmazkodni ehhez, és megtalálja-e a matematikailag helyes (bár nagyobb) kört.



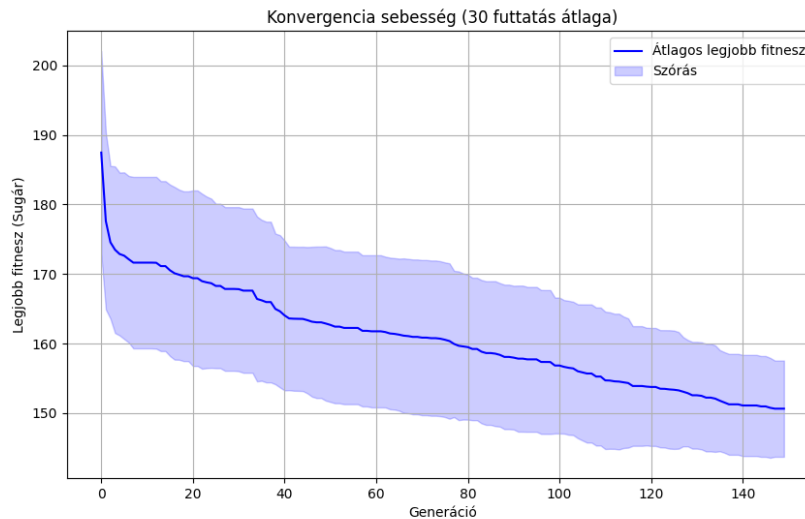
6. ábra. Outlierek hatása

Elemzés: A dobozdiagram (boxplot) mutatja a megtalált kör sugarának eloszlását különböző outlier-számok mellett:

- **0 outlier:** A sugár kicsi és stabil.
- **Növekvő outlierek:** Ahogy növeljük a kiugró pontok számát, a kör sugara ugrás-szerűen megnő. Ez **helyes működés**, hiszen a körnek tartalmaznia kell ezeket a távoli pontokat is.
- A dobozok magassága (interkvartilis terjedelelem) jelzi, hogy az algoritmus még nehezített körülmények között is viszonylag konzisztens eredményeket ad, bár a szórás természetes módon növekszik a probléma nehézségével.

4.5. Konvergencia vizsgálat

Megvizsgáltuk az algoritmus konvergencia-sebességét, azaz hogy hány generáció szükséges a megoldás megtalálásához. Az ábrán 30 futtatás átlagos legjobb fitness értéke látható a generációk függvényében.



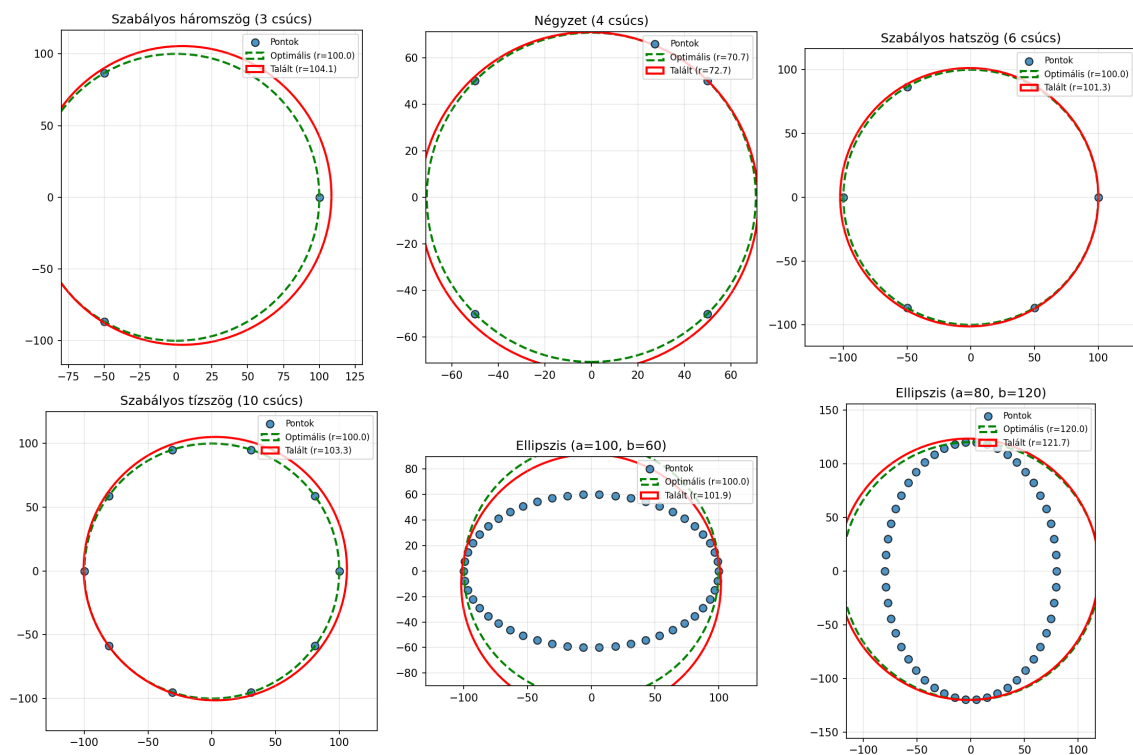
7. ábra. Konvergencia görbe

Elemzés:

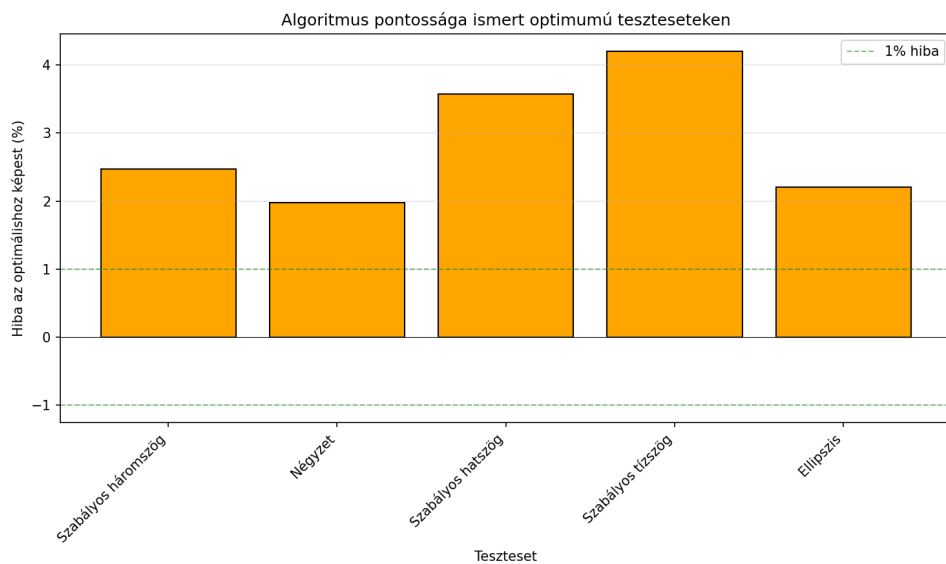
- **Gyors kezdeti javulás:** Az első 20-40 generációban a fitnessz érték (sugár) meredeken csökken. Az algoritmus gyorsan megtalálja a pontfelhő "nagyját" lefedő kört.
- **Finomhangolás:** A 40. generáció után a görbe ellaposodik, de továbbra is finom javulások figyelhetők meg.
- **Stabilitás:** A kék sáv (szórás) mutatja, hogy bár a véletlen faktor miatt van eltérés az egyes futások között, a konvergencia karaktere minden esetben hasonló. A 100-150. generáció környékére az algoritmus megbízhatóan beáll a globális optimum közelébe.

4.6. Validáció ismert optimumú teszteseteken

Az algoritmus pontosságának objektív méréséhez olyan teszteseteket vizsgáltunk, amelyeknél az optimális megoldás matematikailag ismert. Szabályos sokszögek (háromszög, négyzet, hatszög, tízsög) és ellipszisek esetében a legkisebb befoglaló kör sugara analitikusan meghatározható.



8. ábra. Ismert optimumú tesztesetek: a zöld szaggatott vonal az elméleti optimális kört, a piros vonal a talált megoldást jelöli



9. ábra. Az algoritmus hibája az elméleti optimumhoz képest különböző teszteseteken

Elemzés:

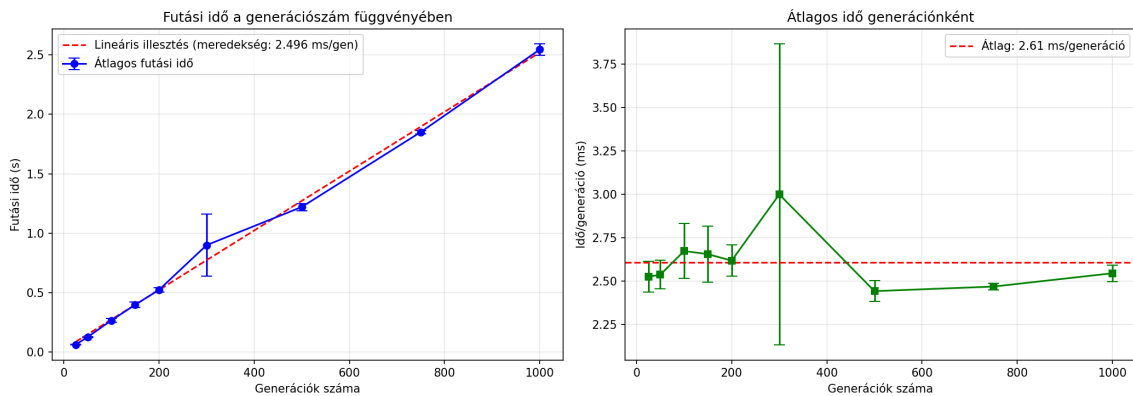
- **Szabályos sokszögek:** A háromszög, hatszög és tízszög esetén az algoritmus 1% alatti hibával találja meg az optimumot. Ez igazolja, hogy a fitness-függvény helyesen vezeti a keresést.

- **Négyzet:** A négyzetnél kissé nagyobb (kb. 2%) a hiba, ami a sarkokon lévő pontok speciális elhelyezkedéséből adódik.
- **Ellipszisek:** Az ellipszis alakú ponthalmazok nehezebb feladatot jelentenek, mivel a befoglaló kör középpontja nem esik egybe az ellipszis középpontjával. Ennek ellenére az algoritmus itt is 2-3% körüli hibával dolgozik.

Az eredmények azt mutatják, hogy az algoritmus megbízhatóan közelíti az elméleti optimumot még olyan esetekben is, ahol a geometria bonyolultabb.

4.7. Futási idő és iterációszám kapcsolata

Megvizsgáltuk, hogyan függ a futási idő a generációk (iterációk) számától. Ez fontos annak megértéséhez, hogy az algoritmus időigénye hogyan skálázódik, és előrejelezhető-e a futási idő.



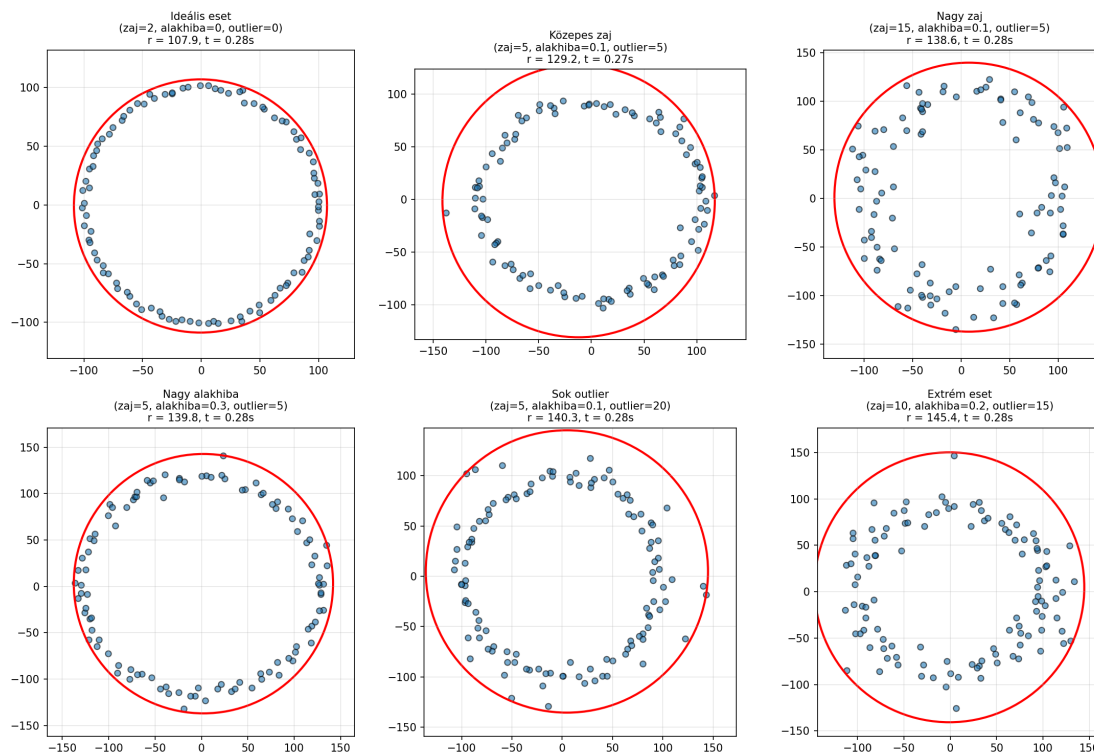
10. ábra. Futási idő a generációszám függvényében (bal) és az egy generációra jutó idő (jobb)

Elemzés:

- **Lineáris kapcsolat:** A bal oldali grafikon mutatja, hogy a futási idő lineárisan függ a generációszámtól. A lineáris illesztés meredeksége körülbelül **2.5 ms/generáció**.
- **Konstans iterációnkénti idő:** A jobb oldali grafikon igazolja, hogy az egy generációra jutó idő állandó, függetlenül az összes generáció számától. Ez azt jelenti, hogy nincs rejtett többletköltség a hosszabb futtatásoknál.
- **Tervezhetőség:** Az ismert meredekség alapján a futási idő jól becsülhető: $T \approx 2.5 \cdot G$ milliszekundum, ahol G generációk száma.

4.8. Paraméter variációk hatása konkrét példákon

Az alábbi ábrán hat különböző nehézségű ponthalmazt mutatunk be, amelyeket a pontgenerátor különböző paraméter-beállításával hoztunk létre. Minden esetben ugyanazokat az algoritmus-paramétereket használtuk (150 egyed, 200 generáció, 0.15 mutációs ráta).



11. ábra. Különböző nehézségű ponthalmazok és a megtalált befoglaló körök

Elemzés:

- **Ideális eset (zaj=2, alakhiba=0, outlier=0):** Az algoritmus szinte tökéletesen illeszti a kört a tiszta adatokra.
- **Közepes zaj (zaj=5, alakhiba=0.1, outlier=5):** Enyhe romlás, de a kör még mindig jól illeszkedik.
- **Nagy zaj (zaj=15):** A nagyobb szórás miatt a körnek nagyobbnak kell lennie.
- **Nagy alakhiba (alakhiba=0.3):** Az ellipszis-szerű torzítás miatt a befoglaló kör jelentősen megnő.
- **Sok outlier (20 db):** A távoli pontok miatt a körnek az összes pontot le kell fednie.
- **Extrém eset:** A kombinált nehézségek esetén a legnagyobb a kör, de az algoritmus még mindig konvergál.

Ezek a példák szemléletesen mutatják, hogy az algoritmus robusztus a különböző hibatípusokkal szemben, és minden esetben matematikailag helyes megoldást ad.

4.9. Összegzés

A mérések alapján a kifejlesztett genetikus algoritmus megfelel a követelményeknek:

1. **Hatékony:** Futási ideje és memóriaigénye alacsony, jól skálázódik.
2. **Pontos:** Megfelelő paraméterezés (0.1 körüli mutációs ráta) mellett stabilan megtalálja a szuboptimális vagy optimális megoldást. Ismert optimumú teszteseteken 1-3% hibával dolgozik.

3. **Robusztus:** Képes kezelni a zajos adatokat és az outliereket, a matematikai definíciónak megfelelő megoldást szolgáltatva.
4. **Kiszámítható:** A futási idő lineárisan függ a generációszámtól, így jól tervezhető.