A brachisztochron probléma megoldása genetikus algoritmus segítségével

Czumbel Péter – ODZF0M

I. Bevezetés

A brachisztochron probléma

Egy függőleges síkon vegyünk fel két pontot, A-t és B-t úgy, hogy ne ugyanazon a függőleges, illetve vízszintes egyenesen helyezkedjenek el. Brachisztochronnak nevezzük azt a pályát, amin a gravitációs mező hatására egy test súrlódás és kezdősebesség nélkül a legrövidebb idő alatt jut el A-ból B-be. A program feladata ennek a görbének a megkeresése adott A és B koordináták esetén.

A genetikus algoritmusok

A genetikus algoritmusok olyan keresőalgoritmusok, amelyek az evolúció, illetve a természetes szelekció lépéseit szimulálva keresnek megoldásokat. A populáció egyedeit itt lehetséges megoldások jelentik, amiket keresztezve új megoldások, új egyedek kaphatóak. Azt, hogy melyik egyed „leszármazottja” kerül tovább a következő generációba, a természetes szelekcióhoz hasonlóan, az egyedek „rátermettsége” határozza meg. A genetikus algoritmusok esetében minél jobb egy megoldás, annál nagyobb lesz az egyed rátermettsége, aminek az értékét fitnessnek szokták nevezni. A fitness értékük alapján kiválasztott egyetek kereszteződnek, mutálódnak, és így új egyedek jönnek létre, amik a populáció következő generációját alkotják. Ideális esetben az algoritmus populációja minden generációval egyre közelebb kerül az optimumhoz.

II. A program működése

A populáció felépítése

A populáció egyedei struktúrákban vannak tárolva, amik tartalmazzák az adott egyedhez tartozó görbe pontjait, a görbén való végigcsúszáshoz szükséges időt és a görbe fitness értékét.

```
typedef struct individ{
    coord* curve;        //a görbe pontjait tartalmazó lista
    double time;         //az idő ami alatt végigcsúszna a görbén egy test
    double fitness;      //a görbéhez tartozó fitness érték
} individ;
```

A „görbe” igazából (x, y) koordináták tömbje, amiket egyenes vonalak kötnek össze.

Az első generáció egyedei random generáltak. A görbe pontjainak X koordinátái adottak, ezek úgy vannak megválasztva, hogy a görbe pontjainak számának megfelelően egyenlő részekre osszák az A és B pont közötti X távolságot. A pontok Y koordinátái viszont random értékek, azzal a feltétellel, hogy az A pont Y koordinátájánál kisebbek. Minden görbe legelső pontja az A pont és legutolsó pontja a B pont. Új egyedeket a következő függvény hoz létre:

```
individ newIndivid(coord A, coord B){
    individ self;           //új egyed létrehozása
    self.curve = (coord*)malloc(sizeof(coord) * (Npoints));
    //memória lefoglalása a görbe pontjainak
    self.curve[0] = A;      //a görbe első pontja A pont lesz
    double max = B.y + 800; //az Y koordináták max 800-al lehetnek kisebbek a B pontnál
    double min = A.y;       //az Y koordináták nem lehetnek nagyobbak az A pontnál
    for(int i = 1; i < Npoints-1; i++){
        self.curve[i].x = A.x + (B.x - A.x)/(Npoints-1) * i;
        //az X koordináták a pontok számával egyenlő részre osztják az x távolságot
        self.curve[i].y = random(min, max);
        //az y koordináták megadása a feltételeknek megfelelően
    }
    self.curve[Npoints-1] = B; //a görbe utolsó pontja B
    self.fitness = 0;   //a görbe fitness érték inicializálása, később lesz kiszámítva
    self.time = -1;     //a görbéhez tartozó idő inicializálása, később lesz kiszámítva
    return self;
}
```

Ezzel a függvénnyel könnyen felépíthetjük a teljes populációt az alábbi függvényt használva:

```
individ* generatePop (coord A, coord B, int popsize, int Npoints){
    individ* pop = (individ*)malloc(sizeof(individ) * popsize);
    //memória lefoglalása a populáció számára
    for(int i = 0; i < popsize; i++)
        pop[i] = newIndivid(A, B);
    //a populáció méretének megfelelő számú egyeddel tölti fel a populációt
    return pop;
}
```

A popsize változó tartalmazza a populáció méretét, az Npoints pedig a görbék pontjainak számát. A fő programrészben így a populáció egyetlen hívással létrehozható:

```
individ* pop = generatePop(A, B, popsize, Npoints);
```

Egy generáció kiértékelése

A kiértékelés során mindegyik egyednek ki kell számítani a fitness értékét, amihez először a pályán való végig haladás idejét kell kiszámítani. Mivel a görbe szakaszai egyenesek, egy szakaszon számolhatunk a test mozgásával úgy, mintha egy lejtőn csúszna. Mivel ismerjük a lejtők kezdő és végpontjait, könnyen kiszámíthatjuk a magasságát, hosszát és a szögét, egyedül a kezdősebességeket nem tudjuk. Ez viszont mindig megegyezik az előző szakasz végsebességével, amit szintén kiszámolható, mert az első szakasz kezdősebessége 0.

$$s = \sqrt{\Delta x^2 + \Delta y^2} \qquad a = \frac{\Delta y}{s} \cdot g \qquad t = \frac{-v + \sqrt{v^2 + 2aS}}{a} \qquad v1 = v0 + a * t$$

Mindezt a következő függvény számítja ki:

```
double calcFitness(individ* self, coord A, coord B){
    double time = 0;     //az idő inicializálása
    double speed = 0;    //kezdősebesség inicializálása
    double dx = (B.x - A.x)/(Npoints);
    //a görbe két pontja közötti X távolság kiszámítása
    for (int i = 1; i < Npoints; i++){
        double dy = self->curve[i].y - self->curve[i-1].y;
        //a görbe két pontja közötti Y távolság kiszámítása
        double s = sqrt(dx*dx + dy*dy);
        //a görbe két pontja közötti teljes távolság kiszámítása
        double a = dy/s * g;
        //a csúszó test gyorsulásának kiszámítása az adott szakaszon
        double t = (-speed + sqrt(speed*speed - 4 * a/2 * -s)) / (2 * a/2);
        //az adott szakaszon való végigcsúszás idejének kiszámítása
        time += t;
        //a szakasz idejének hozzáadása a teljes időhöz
        speed = speed + a * t;
        //a következő szakasz kezdősebességének kiszámítása
    }
    double fitness = 1/pow(time, 3);
    //a fitness érték a teljes idő -3-adik hatánya így minél
    //nagyobb fitness érték minél jobb görbét jelent és kis
    //időbeli különbség nagyobb eltérést eredményez a fitnessben
    if (fitness != fitness){  //ha a fitness érték nem valós szám,
        self->time = -1;      //mert a görbén nem tudna végighaladni a test
        return 0;             //-1-re állítja az időt, és a fitness 0 lesz
    } else {
        self->time = time;    //beállítja a görbe idejét a kiszámított értékre
        return fitness;       //visszatér a fitness értékével
    }
}
```

A görbéknek adott fitness a kiszámított idő -3-adik hatványa. Ez azért hasznos, mert így a jobb görbékhez nagyobb fitness tartozik, valamint egy kis előny is nagyban javítja egy egyed esélyét, hogy egy leszármazottja bekerüljön a következő generációba.

A fenti függvény elméletileg bármilyen pozitív értéked adhatna, ezért minden egyed fitnessét elosztjuk az adott generáció legnagyobb fitness értékével, hogy csak 0 és 1 közötti számokkal kelljen dolgoznunk.

```
void evaluate(individ* pop, coord A, coord B){
    for(int i = 0; i<popsize; i++)
        pop[i].fitness = calcFitness(&pop[i], A, B);
    //a fitness kiértékelése az összes egyedre
    double maxFitness = calcMaxFitness(pop);
    //a maximum fitness kiszámítása
    for(int i = 0; i<popsize; i++)
        pop[i].fitness /= maxFitness;
    //a fitness értékek normalizálása, 0 és 1 közötti értékekre
}
```

A fenti függvény kiszámítja a populáció összes egyedének fitnessét és normalizálja azokat.

A szaporodó egyedek kiválasztása

Azt, hogy melyik egyén „génjei” adódnak tovább, az egyed fitnesse dönti el. Minél nagyobb egy egyed fitnesse, annál nagyobb eséllyel kell szaporodnia.

```
individ select(individ* pop, double maxFitness){
    int failsafe = 0; //ha túllépi a 10000-et, továbbengedi a programot
    while (true){
        int i = rand()%popsize; //kiválaszt egy random egyedet
        double r = random(0, maxFitness); //kiválaszt egy random fitness értéket
        individ self = pop[i];
        if (r < self.fitness || failsafe > 10000)
            return self;
        //Ha a kiválasztott egyed fitnesse nagyobb mint
        //a random érték, az egyed ki lessz választva.
        //Ha kisebb, új egyedet választunk.
        failsafe++;
    }
}
```

A fenti szelekciós függvény ezt úgy valósítja meg, hogy először random választ egy egyedet és generál hozzá egy random számot 0 és 1 között. A választott egyed csak akkor lesz ténylegesen kiválasztva, ha a fitnesse nagyobb, mint a random generált érték. Ez azt jelenti, hogy mindegyik egyednek van esélye a szaporodásra, de ez az esély arányos a fitnessével.

Két egyed keresztezése

Akárcsak az élőlények esetében, itt is két egyed génjeiből áll össze a leszármazott. Biológiai megkötések ugyan nincsenek, ezért bárhány egyedet keresztezhetnénk, de kettővel a legegyszerűbb dolgozni. A keresztező algoritmus mindössze annyit csinál, hogy kiválaszt egy pontot a leszármazott génjében. A pont előtt az egyik szülő génjeit kapja, a pont után pedig a másikét. A géneknek ebben az esetben a görbék pontjai felelnek meg.

```
individ crossover(individ parentA, individ parentB, coord A, coord B){
    individ child = newIndivid(A, B);    //leszármazott létrehozása
    int midpoint = random(0, Npoints);  //a törés helyének kiválasztása
    for(int i = 0; i < Npoints; i++){
        if(i < midpoint){
            child.curve[i] = parentA.curve[i];
            //a kiválasztott hely előtti gének bemásolása A szülőből
        } else {
            child.curve[i] = parentB.curve[i];
            //a kiválasztott hely utáni gének bemásolása B szülőből
        }
    }
    return child;
}
```

Mutációk

A mutációkra a populáció diverzitásának fenntartásához van szükség. Mutációk nélkül egy idő után mindegyik egyed egyforma lenne, és az algoritmus elakadna.

```
individ mutate(individ self, coord A, coord B){
    //pontmutáció
    for(int i = 1; i < Npoints-1; i++){
        if(random(0, 1) < PointMutationRate){
            double max = (B.y-A.y)/Npoints;
            self.curve[i].y += random(-max, max);
            //a mutációs aránynak megfelelő eséllyel
            //random értékkel változik az adott pont
        }
    }
    //hosszúmutáció
    if(random(0, 1) < LongMutationChance){
        int a = (int)random(1, Npoints-1-5);
        int b = (int)random(a+5, Npoints-1);
        double r = random(-30, 30);
        for(int i = a; i < b; i++){
            self.curve[i].y += r;
        }
        //random kiválasztott A és B pont között az
        //összes pont azonos, random értékkel változik
    }
    return self;
}
```

A mutációs függvény kétfajta mutációt alkalmaz. Először végig lepkéd a kapott egyed génjein, és adott eséllyel egy random értékkel megváltoztatja azt a gént, ez a pontmutáció. A mutációs esély 1-2% körül adta a legjobb eredményeket.

A második fajtánál a függvény kiválasztja két pontját a génnek, és a két pont között minden értéket azonosan változtat meg. Ez talán a kromoszómamutációk megfelelője lehetne a természetbe. Erre a fajtára azért van szükség, mert a pontmutációkkal kisimulhat a görbe, de utána minden újabb pontmutáció egy tüskét okozna benne, ami nagyban csökkentené a fitnessét. A hosszú mutációk tüskék létrehozása nélkül is tudják mozgatni a görbe pontjait.

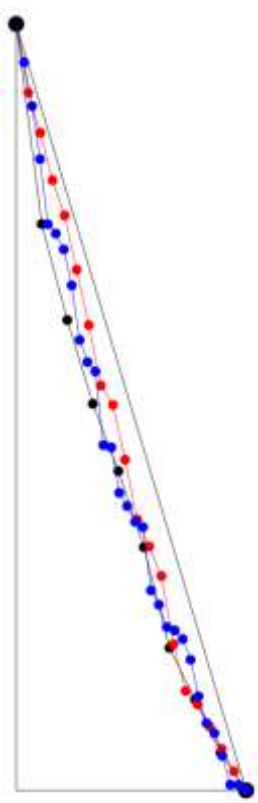
A főprogram

A fő programrész lényeges része:

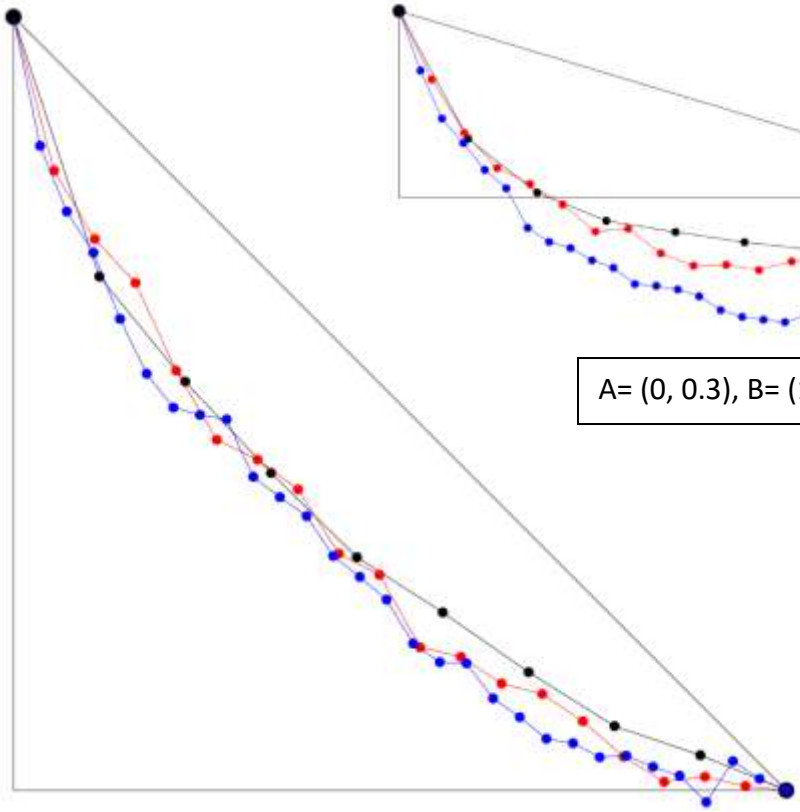
```
while(currentGen < maxGen){
    currentGen += 1;
    evaluate(pop, A, B);
    //populáció kiértékelése
    individ* newGen = (individ*)malloc(sizeof(individ) * popsize);
    //memória lefoglalása az új generációnak
    double maxFitness = calcMaxFitness(pop);
    for(int i = 0; i < popsize; i++)
        newGen[i] = mutate(crossover(select(pop, maxFitness), select(pop, maxFitness),
                    A, B), A, B);
    //az új generáció feltöltése a leszármazottakkal
    best = writeToFile(pop, "graph.csv");
    for(int i = 0; i < popsize; i++)
        free(pop[i].curve);
    //előző generáció memóriájának felszabadítása
    free(pop);
    //a popluáció memóriájának felszabadítása
    pop = newGen;
    //az új generáció lesz a populáció
}
```

A főprogram addig készít új generációkat, amíg el nem éri a megadott generációszámot. A program a futása végén a legjobb pályából egy SVG rajzot generál, valamint van lehetőség minden generáció után kiírni egy fájlba a generáció átlagos idejét és fitness értékét.

III. A generált adatok értékelése



A= (0, 1), B= (0.3, 0)

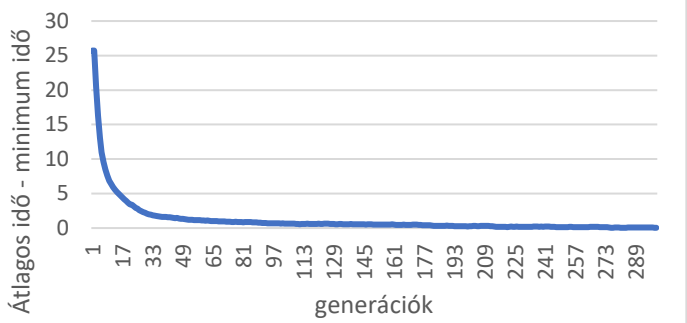


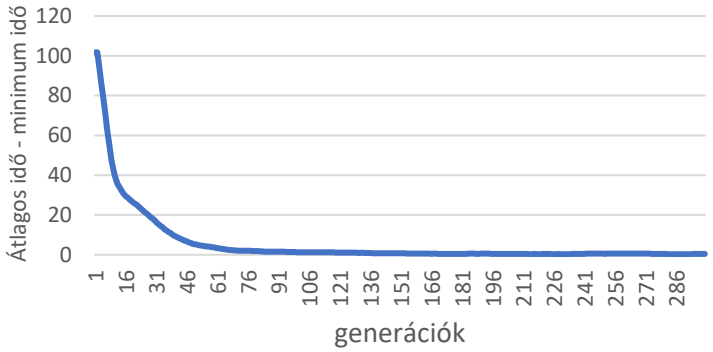
A= (0, 1), B= (1, 0)

A= (0, 0.3), B= (1, 0)

A fenti három ábra 5000-es populációszámmal 5000 generáció után mutatja meg az algoritmus által talált legjobb megoldásokat. A fekete görbéken 10, a pirosokon 20, a kékeken 30 pont alkotja a görbéket. Általános tapasztalatom volt, hogy kevesebb pontot használva sokkal nagyobb valószínűséggel talált az algoritmus egy közel ideális megoldást.

A jobboldali ábrán látható leginkább, hogy mi történik sok pont esetén: Az algoritmus talál egy viszonylag sima, de az ideálisnál alacsonyabb görbét, és annak az esélye, hogy egy mutáció javítani tud azon a görbén, nagyon kicsi. Ezen lehet, hogy több generációval lehetne segíteni, de sok pont esetén a program kifejezetten lassan fut, 5000 generációval is kb. 2 perc volt amíg lefutott a számítógépemen.





A fenti grafikonok alapján is úgy tűnik, pár száz generáció után jelentős javulás már nem történik a populációban, a diverzitás annyira lecsökken, hogy nagy változások nem következnek be. A mutációk esélyének növelésével sem érünk el sok mindent, egy bizonyos szint fölött olyan eredményeket ad az algoritmus, mitnha minden generációt random generáltunk volna. Ez azért történik, mert ha egy görbén mutáció történik, az nagy valószínűséggel ront a görbén, és az a következő generációból kiszelektálódik. Éppen ezért a mutációk csak egy bizonyos fokig tudják növelni a populáció diverzitását, afölött pedig már rontanak az eredményen.

A diverzitás lecsökkenését meg lehetne akadályozni azzal, ha egymáshoz nagyon hasonló egyedeket nem engednénk be a következő generációba, ezzel megakadályozva, hogy a program egy rosszabb, de sima görbén akadjon meg, de ennek a programrésznek az elkészítése már nem fért bele projektbe.

Czumbel Péter – 2021.01.24.