Diplomamunka

Készítette:

Ceglédi Róbert

Debrecen

2009

Debreceni Egyetem

Informatika Kar

Fejlett kereső algoritmusok Java-ban

Témavezető:
Jeszenszky Péter
adjunktus

Készítette:
Ceglédi Róbert
programtervező matematikus

Debrecen

2009

Tartalom

1. Bevezetés

A mesterséges intelligencia a számítástudomány talán legjelentősebb és legkiterjedtebb ága. Feladata, hogy az emberi tudatos viselkedéshez hasonlóan tudjon feladatokat megoldani. Segítségével olyan valós életbeli problémákra adhatunk választ, melyeknek a feltárásához nem elegendőek csak matematikai eszközök, hanem olyan intuitív technikákat kell alkalmazni, melyeket nehéz formalizálni, algoritmizálni. Talán érthető, hogy e problémák és algoritmusok java miért váratott magára a számítástechnika megjelenéséig. A mesterséges intelligencia keresőalgoritmusaira jellemző, hogy a megoldást igen nagy halmazból kell kiválasztaniuk, és az egyes megoldásjelöltekről nem mindig lehet megállapítani, hogy milyen távol vannak a globális optimális megoldástól. A tudományág alig több mint fél évszázada létezik, John McCarthy használta először a mesterséges intelligencia kifejezést 1950-ben. Ilyen problémákkal a számítógépek megjelenése előtt számítási erőforrás hiányában viszonylag kevesen foglalkoztak, habár a megoldó algoritmusok egy része már a számítógépek megjelenése előtt létezett. Ma már a tudomány szinte minden területén jelen van valamilyen formában (orvostudomány, képfeldolgozás, katonai alkalmazások, robotika, stb.) és az évek haladtával mind egyre jobban a napjaink részévé válik.

Munkám során olyan fejlett kereső algoritmusok után kutattam, melyek az életben is előforduló, diszkrét matematikai problémákra adnak választ. Célom egyrészt az volt, hogy átfogó képet írjak le a témakörből, hogy milyen technikák, eljárások és módszerek léteznek ezen a területen, másrészt a szerintem legintuitívabb részterületét, az evolúciós jellegű algoritmusokat szerettem volna jobban bemutatni. Ezen módszertant a legtöbb hazai irodalom az utazó ügynök problémájának (angol rövidítéssel: TSP) segítségével tárgyalja, én sem döntöttem másként. Kidolgoztam egy evolúciós problémamegoldó keretrendszert, mely segítségével könnyen lehet implementálni az ilyen jellegű algoritmusokat és problémákat. A program Java nyelven készült, az utazó ügynök problémát és megoldását grafikusan is meg tudja jeleníteni. Az eredmények ellenőrzéséhez a Concorde [9] nevű programot használtam fel, mely segítségével elő lehet állítani az optimális körutat. Megkísérlem összegyűjteni, hogy melyek azok a nem csak evolúciós algoritmusok, melyek az erőforrásigény vagy az idő függvényében egy elég jó megoldást adnak erre a problémára. Több mérési tesztet végzek el, és ezek alapján felállítok egy sorrendiséget az algoritmusok hatékonyságának függvényében.

2. Számítási intelligencia

A számítási intelligencia (Computational Intelligence) a mesterséges intelligencia olyan ága, ahol a problémamegoldás lépésenkénti fejlődésen vagy tanuláson keresztül hajtódik végre. A számítási intelligenciát alapvetően három jól elkülöníthető területre bonthatjuk szét, melyekre a következő fejezetek rálátás-szintű áttekintést nyújtanak.

2.1. Neurális hálók

Az evolúciós fejlődés csúcsaként az embert szokták megnevezni. Ez érthető, hiszen egy olyan komplex rendszer irányítja, melynek kifejlesztéséhez több millió évre volt szükség: az emberi agy. Térfogata megközelíti az 1500 cm^3-t és mégis gyökeresen képes volt megváltoztatni a világ arculatát. Lényegében egyszerű információ-feldolgozó egységekből, neuronokból áll, melyek szinapszisok segítségével kapcsolódnak össze és kommunikálnak egymással.

A kutatók ennek mintájára egy robusztus, nagyon jó hibatűrő modellt alkottak, mely leginkább olyan területeken használatosak, ahol a probléma megoldására többnyire nincs algoritmus vagy szabályrendszer, rengeteg paramétertől függ és ezek a paraméterek esetleg hibával terheltek. Olyan számítási modellről van szó, mely biológiai eredetű, de ezektől már elszakadt. A neurális háló sok homogén, egyszerű, azonos működésű elemből állnak, ezek a mesterséges neuronok. A neuron úgy határozza meg a kimenetét, hogy a bemeneti súlyok lineáris kombinációját képzi, majd ezt átadja az aktivációs függvénynek ami a kimenetet szolgáltatja. Minden neuron a saját állapotának megfelelően generál egy kimeneti értéket, melyet a vele kapcsolatban álló neuronokkal közöl. A leggyakrabban alkalmazott architektúra az előrecsatolt hálózat, ahol a neuronok a következő rétegekbe szerveződnek:

- Input réteg: neuronjai kapják a bemeneti paramétereket, a problémát leíró adatokat
- Rejtett réteg(ek): az input rétegtől kapott adatokat dolgozzák fel a fent említett módon
- Output réteg: neuronjai szolgáltatják a hálózat kimenetét.

A rétegek közötti kapcsolatok egyirányúak, az adatok az input rétegtől kiindulva az output réteg felé haladnak.

Egy neurális háló programozása nem szekvenciális utasításokkal történik, hanem úgynevezett tanító folyamat segítségével változtatják az egyes kapcsolatokhoz tartozó súlyokat, ezzel hatást gyakorolva az output réteg által szolgáltatott megoldásra. Annyit kell csak tudunk a problémáról, hogy az egyes bemenetekre milyen kimeneti értékeket kell kapnunk, majd egy iterációs eljárással a súlyokat korrigáljuk, tanítjuk, hogy az output réteg által szolgáltatott megoldás a lehető legjobban igazodjon a várt értékekhez.

2.2. Fuzzy rendszerek

„Lotfi Zadeh szerint az emberi gondolkodásmód sokkal jobban modellezhető olyan fogalmakkal, amelyeknek nincsenek éles határaik, ahol az átmenet egy tulajdonság megléte és nem megléte között homályos (angolul: fuzzy)” [7]. Tehát a fuzzy rendszerek olyan feladatok megoldására valók, melyek fogalmait nem tudjuk hagyományos egzakt matematikai formulákkal kifejezni, hiszen azok túl merevek és pontosak lennének, minden bizonytalanságot igyekeznek kizárni a definíciókból. Például ha egy adatbázisból nagy csomagterű autót keresünk, akkor a hagyományos rendszereknél pontosan meg kellene adni, hogy pontosan mi számít nagy csomagtérnek és milyen paraméterei vannak egy városi autónak.

A fuzzy rendszerek a többértékű logikán alapulnak, azaz egy formula kiértékelése nem csak igaz vagy hamis, hanem ennél általánosabb, az igazságértékek a [0,1] intervallumból bármely értéket felvehetik, tehát azok spektrumáról beszélünk. Nagyon jól alkalmazható például a mikro-kontrollereknél, a bonyolult automatizált eljárású szabályzó rendszereknél vagy a mesterséges intelligencia területén egyaránt.

2.3. Evolúciós technikák

Olyan számítási technikáról van szó, melynek kifejlesztéséhez a természet adott ihletet. A törzsfejlődés során, ahogyan az egyes fajok kialakultak a természetes kiválasztódás törvénye alapján az évmilliók alatt, úgy lehet párhuzamot vonni a számos lehetséges megoldások kiválasztása között az egyes iterációs lépésekben. A végcél nem egy jól körülírt állapot, hanem az ismert potenciális megoldások között kiválasztani a számunkra legmegfelelőbbet. Lényegében egy kezdeti, lehetséges megoldásokat tartalmazó halmaz

elemein végzett egyszerű műveletek segítségével olyan eredményhalmazt kapunk, melynek elemei közel állnak a várt megoldáshoz. Minden problémára külön versenyeztetési és kiválasztási eljárások léteznek, melyekről részletesebben a következő fejezetekben lesz szó.

3. Evolúciós modell a programozásban

3.1. Ismertetés

A kizárólag egzakt matematikai módszerekre épülő feladatmegoldó rendszerek egyik hátrányos sajátossága a rugalmatlanság. Ahányféle különböző problémával találták szembe magukat a matematikusok, annyiféleképpen lehetett megoldaniuk az adott feladatot, ezért minden problémához különböző képleteket, egyenleteket és matematikai eljárásokat kellett kidolgozni, melyek a feladat komplexitásának függvényében akár évekig is eltarthattak. Más-más módszerekkel dolgoznak például az operációkutatás, a statisztika vagy a numerikus analízis területén is. Olyan innovatív módszerre volt szükség, mely ezeket a hátrányosságokat áthidalja és közelebb hozza egymáshoz a problémákat. A legjobb megoldást talán a természet nyújtja, hiszen olyan technológiával dolgozik, mely az univerzum időskálájának egy igen kis hányada alatt nagyon bonyolult és összetett hibatűrő rendszereket hozott létre. A dolog szépsége az, hogy egyetlen központi törvényszerűség jellemzi a folyamatot: a természetes kiválasztódás.

Charles Darwin „A fajok eredete” (1859) című munkája ihlette a matematikusokat, hogy olyan adaptív rendszereket fejlesszenek, melyek egy magasabb absztrakciós szinten általános módszert nyújtanak akár egymástól teljesen eltérő problémák megoldásában is. Az alapötlet az, hogy a természet elemeit olyan egyedek alkotják, melyek folyton versengésben állnak egymással és ezáltal az adott fajtól függően bizonyos tulajdonság szerint különbséget lehet tenni közöttük az „alkalmasság” tekintetében. A jobb, tehetségesebb egyedek fennmaradási esélye nagyobb, hatékonyabban veszik fel a harcot a populáció többi tagjával szemben. Az ökölszabály az, hogy az erősebb fennmarad, a gyengébb elbukik. Mivel az öröklődés során az egyedek magukban hordozzák a szüleik tulajdonságait, – és ezt csak az alkalmasnak bizonyult egyedek tehetik meg – az idő haladtával egyre erősebb populáció jön létre. A generációk haladtával a versengés egyre kiélezettebbé válik, mígnem egy magasabb rendű faj jön létre, tagjai átlagosan bizonyos szempontból közel állnak az ideális állapothoz. A problémamegoldásban, az algoritmusok tervezésénél ezt a modellt nagyon hatékonyan fel lehet használni, ezt már a számítástechnika hajnalán, az 50-es években felismerték. Az idő haladtával egyre szélesebb körben tudták kiterjeszteni az egyes tudományterületekre, a 60-es évek végén kísérleti jelleggel szinte már minden tudományág alkalmazta a saját evolúciós

problémamegoldó algoritmusait.

3.2. Alapfogalmak

A probléma lehetséges megoldásait **egyedeknek vagy individuumoknak** (individual) nevezzük, melyek génekből épülnek fel. Ezek mindegyikét elfogadhatjuk a feladat végeredményeként, de éppen az a célunk, hogy ezek közül a legjobbat válasszuk ki. A köztük lévő sorrendiség felállításáért egy úgynevezett **rátermettségi (fitness) vagy alkalmassági függvény** felel. A magasabb fitness érték a feladat szempontjából jobb tulajdonságú egyedre utal. Az egyedeknek kezdetben egy kisebb részhalmaza van jelen, ez a kiinduló **populáció.** Elemeit tipikusan véletlenszerűen állítják elő a megoldó algoritmusok, ezek nagy valószínűséggel távol állnak az optimális megoldástól. Az előállításánál nem szempont, hogy különböző elemeket tartalmazzon, megengedett a többszörös duplikáció is. Az új egyedek létrehozása a **keresztezés** feladata, mely a populáció egyediből oly módon hozza létre őket, hogy azok lehetőleg előnyösebb tulajdonságát ötvözi valamilyen stratégia szerint. Néhány irodalom a keresztezés fogalmát csak akkor használja, amikor az utódok előállítására legalább két őst használnak fel. Ezzel ellentétben én kiterjedtebben használom ezt a fogalmat. Opcionálisan szoktak alkalmazni **mutációt,** mely bizonyos fokú véletlenszerűséget visz az új egyedek létrehozásába. A populáció mérete véges, a gyengébb egyedeket egy **szelekciós operátor** segítségével ki kell selejtezni a rátermettségi függvény alapján. A populációk iterációs lépések során megalkotott sorozatát **generációknak** nevezzük, mely az idő haladtával várhatólag egyre erősebb populációkat fog tartalmazni. Az algoritmus futásának végén az utolsó generáció legjobb egyedét tekintjük a feladat megoldásának. A végcél nem mindig egzakt, legtöbb esetben nem az a jellemző, hogy pontosan meg van adva, hogy mely egyedet keressük, hanem körülírjuk, hogy milyen tulajdonságoknak kell eleget tenniük.

3.3. Az evolúciós algoritmusok általános működése

3.3.1. Egyed létrehozása

Első lépésként pontosan meg kell fogalmazni az adott problémát, melyből a tulajdonságok megfogalmazásával származtatni tudjuk az egyedeket. Rögzíteni kell, hogy a tulajdonságok milyen értéket vehetnek fel, egymással milyen viszonyban kell lenniük ahhoz, hogy potenciális megoldásként tekinthessünk rájuk. Ki kell választani azokat a jellemzőket,

amelyek alapján versenyeztetni akarjuk az egyedeket, ezeket általában számadatokkal implementáljuk. Ezután tudjuk megalkotni az alkalmassági függvényt, mely kiszámolja, hogy az adott egyed az adott tulajdonságok alapján mennyire életképes, hogyan viszonyul a társaihoz.

3.3.2. Populáció létrehozása

A kezdeti populáció megalkotásához fel kell azt tölteni véletlen egyedekkel. A populáció számosságát vagy előre rögzítjük, vagy a rátermettségi függvény alapján szabjuk meg, hogy a populáció mely egyedei alkothatják. Utóbbi esetben ez generációról generációra változhat. Minél több társával tud egyszerre versenyre kelni egy egyed, annál valószínűbb, hogy csak a jobb tulajdonsággal rendelkezők maradnak életben. Az algoritmusok a kezdeti populáció egyedeit valamilyen stratégia szerint kiválogatják, majd keresztezik őket egymással. Ezek történhet heurisztikusan, szisztematikusan vagy egyszerűen csak véletlenszerűen. Bármelyiket is alkalmazzuk, a kiválasztódás gondoskodik arról, hogy ha rosszabb egyedek is jönnek így létre, azok a következő generációban ne vegyenek részt, vagyis a szelektáló eljárás segítségével terminálja a rosszabb egyedeket. Fontos úgy megalkotni őket, hogy az így előállított leszármazottak továbbra is szabályos potenciális megoldások legyenek, azaz az egyedekre a konzisztenciát mindig biztosítani kell. Feladattól függően több keresztezési eljárás is létezik, de mindegyik arra hivatott, hogy azon tulajdonságokat emeljék ki a szülőből, amelyek örökítésével a leszármazott(ak) legalább olyan alkalmasak lesznek mint ők. Mindegyik módszernek megvan a maga pro és kontra jellemzője: például az irányítottság az egyedek kijelölésénél a keresztezéshez bizonyos fokú determinisztikusságot eredményez, holott abban rejlik az evolúciós algoritmusok ereje, hogy olyan irányba terelik a problémamegoldást, amely azelőtt feltáratlan terület volt és talán épp ott van a legjobb tulajdonságokkal rendelkező egyed. Szélsőséges esetben ez annyira leszűkítheti a bejárt megoldásteret, hogy bizonyos generációkban a populációk nagyon hasonló egyedeket tartalmaznak. Ez az úgynevezett **lokális optimum** körüli beragadás**,** a matematikában a **gradiens módszernek** ez az egyik sajátossága. Erre nyújt megoldást az úgynevezett **mutáció,** mely bizonyos fokú véletlenszerűséget visz a problémamegoldásba. Általában a keresztezés közben vagy után szokták alkalmazni, de ezekre nincs különösebb megkötés, egyedül a konzisztencia betartására kell odafigyelni.

3.3.3. Szelekció

A szelekciós eljárás a populáció egyedeire alkalmazott keresztezési algoritmus lefutása után lép életbe. Többféleképpen is meg lehet oldani, de általános működése a következő: a régi populációt és az újonnan létrejött individuumokat egy halmazként tekinti, majd ezek közül a következő generáció számára csak azokat tartja meg, amelyek a rátermettségi függvényük alapján megfelelőnek bizonyultak. Mindig az előre beállított populáció méretéhez igazodik. Vannak olyan szisztematikus keresztező eljárások, melyek az új egyed létrejötte után rögtön ítélkeznek, hogy alkalmasabb-e a szülőnél. Ha igen, akkor az a szülő helyébe lép, más esetben pedig a spártai módszerhez hasonlóan rögtön „terminálódik" és nem vesz részt a további műveletekben.

3.3.4. Terminálás

Az utolsó generáció tartalmazza a feladat egy lehetséges megoldását, ez persze nem biztos, hogy a legoptimálisabb lesz. Azt, hogy mikor érjen véget az algoritmus, kétféleképpen lehet meghatározni:

- ha előállította az előírt mennyiségű generációkat,
- vagy ha az aktuális generáció legjobb egyede(i) megfelelnek az adott terminálási feltételeknek.

Az egyes algoritmusok eltérő hatékonyságából kifolyólag a generációk maximális száma változik. Egy bizonyos idő után már nem érdemes tovább futtatni a probléma-megoldást, mert az aktuális populáció egyedei annyira közel állnak egy lokálisan optimális megoldáshoz, hogy nehéz lenne újabb egyedeket előállítani, az pedig valószínűleg alig térne el az ideálistól. Ajánlott beépíteni az algoritmusokba egy ellenőrzést, hogy ha pár generáció után a populációk egyedeinek jósága alig változik, akkor az terminálódjon.
Az evolúciós algoritmusok általános működése a következő:

1. populáció := véletlen_populáció_generálása
2. előállított_generációk_száma := 0
3. **while** (előállított_generációk_száma < generációk_maximális_száma) **do**
4. szülők:=szülők kiválasztása(populáció)
5. új_egyedek := keresztezési_eljárás(szülők)
6. új_egyedek := mutáció(új_egyedek)
7. összes_egyed := populáció + új_egyedek

8. populáció := szelekciós_eljárás(összes_egyed)
9. **end while**
10. **return** legjobb_egyed(populáció)

4. Az evolúciós keretrendszer ismertetése

4.1. Alap interfészek

A keretrendszer megírása Java nyelven történt. Az egyedek implementálásához találni kell egy hatékony adatszerkezetet, mely az egyed állapotát reprezentálja. Ez mindig feladatfüggő, ezért az absztrakciós szinten nem jelenik meg. Két tulajdonságot emeltem ki, mellyel általánosan minden egyednek rendelkeznie kell:

- alkalmassági (fitness) érték megállapítása
- véletlen példány előállítása.

Az alkalmassági érték a jóságértékét szinonimája, minden individuumnak egyedileg kell implementálnia. Ha két egyed állapota különbözik, akkor egyértelműen el kell tudni dönteni, hogy melyikük alkalmasabb, ez sorrendiséget is definiál. Éppen ezért szükséges a `Comparable` interfész implementálása. A véletlen példányok előállítására a kezdeti populáció inicializálásakor van szükség, ez pedig egyed-specifikus művelet. Az ezt megvalósító `Individual` interfész a következőképpen nézki:

```
package framework;

/**
 * Evolúciós problémák leírásához való interfész.
 */
public interface Individual extends Comparable{

    /**
     * Az evolúciós probléma olyan tulajdonsága, mellyel
     * jóság-érték szerint különbséget lehet tenni közöttük.
     * @return az egyed mérőszáma
     */
    public double fitness();

    /**
     * Véletlen példányokat hoz létre
     * @return véletlenszerűen előállított egyed
     */
    public Individual createRandomIndividual();
}
```

Az evolúciós algoritmusok közös jellemzőit nehezebben lehet összefoglalni, az egyik lényeges tulajdonságuk, hogy ha megkapják a probléma reprezentációját és a populációt, akkor abból elő tuják állítani az új egyedeket. Az `EvolutionaryAlgorithm` osztály tartalma a következő:

```
package framework;
import java.util.Vector;

/**
 * Evolúciós algoritmusok absztrakt osztálya.
 */
public interface EvolutionaryAlgorithm {

     /**
      * Minden evolúciós problémát megoldó algoritmusnak annyi
      * a feladata, hogy a paraméterként kapott evolúciós
      * probléma alapján előállítsa a következő generáció
      * egyedeit.
      * @param problem evolúciós probléma
      * @return a következő generáció egyedei
      */
      public Vector<Individual> nextGeneration(
          EvolutionaryProblem problem );
}
```

A `nextGeneration` függvény bemenete egy `EvolutionaryProblem` típusú objektum, mely tartalmazza a populáció egyedeit, valamint a problémamegoldó algoritmus egy példányát. Eredményül elegendő az új generáció egyedeit visszaadni.

A problémamegoldás menete lehet sokféle, de bármelyik módszert is választjuk, legyen az evolúciós vagy neurális technika, egy megoldásnak szánt példányt vissza kell adniuk. Evolúciós esetben ez egy `Individual` leszármazott lesz, az ezt rögzítő interfészt pedig a `Solver.java` fájl tartalmazza:

```
package framework;

/**
 * A problémamegoldó algoritmusok közös viselkedését leíró
 * interfész, legfelsőbb absztrakciós szint. Ezen
 * keretrendszeren belül az evolúciós és az általános megoldó
 * algoritmusok terjesztik ki. */
```

```
public interface Solver {

     /**
      * Az algoritmus által kezelt probléma megoldása.
      * @return egyed, a probléma egy megoldása
      */
     public Individual solution();

     /**
      * Információkat közöl a program futásáról, a megoldó
      * algoritmusok szabadon implementálhatják.
      * @return információk
      */
     public  String stats();

}
```

A `solution()` metódus visszatérési értéke szolgáltatja majd a probléma megoldását. Az `stats()` metódus csupán információs célokat szolgál, nem elengedhetetlen része a keretrendszer működésének. Visszatérési értéke egy olyan `String`, mely információkat tartalmaz az egyes generációkról, például a kezdeti és az utoljára előállított populáció legjobb egyedének fitness értékét, ezek százalékos eltérését, valamint a futási időt. A későbbiekben majd láthatunk néhány példát is erre.

A nem evolúciós jellegű algoritmusok implementálásához a `NonEvolutionary` interfészt kell implementálni, mely a `Solver` interfész kiterjesztése:

```
package framework;

/**
 * Általános megoldó algoritmusok interfésze.
 *
 */
public interface NonEvolutionary extends Solver {

     public void solve(Individual tsp);
}
```

Az interfész egyszerűen a célt szolgál, hogy segítségével a keretrendszer el tudja indítani a keresést, ehhez a `solve(Individual tsp)` metódust kell definiálni,

paramétereként a problémát reprezentáló objektumot kell átadni. A metódusnak, mint látható nincs visszatérési értéke, a megoldást a `Solver` interfész `solution()` metódusa fogja szolgáltatni.

4.2. Csomagoló osztályok

Az egyed jellemzőit rögzítő osztály megalkotása után be kell azt építeni a keretrendszerbe, erre írtam meg az `EvolutionaryProblem` osztályt. A konstruktorának egy `Individual` interfészt implementált példányt kell átadni, valamint az előre meghatározott generációk felső határát és a populáció maximális méretét. Ez utóbbi két adatot a program indítása során paraméterként kell megadni. A populáció inicializálása is itt történik, egy ciklusban meghívódik az egyed kötelezően implementált, véletlen egyedek előállítására szolgáló `createRandomIndividual()` metódusa. Az `EvolutionaryProblem` osztály következő néhány sora ezt végzi el:

```
private Vector<Individual> population;
private Individual individualProblem;

/**
 * Populáció feltöltése véletlen egyedekkel
 */
private void createPopulation() {
     for (int i=0;i<populationSize;i++)
          population.add(
          individualProblem.createRandomIndividual());
}
```

Amint az látható, a populáció mérete előre rögzített. Az `individualProblem` referencia-változó tartalmazza azt az objektumot, melyet a konstruktor egyik paramétereként adunk át, tartalmazván az egyed példányát. Ezzel már megfelelően van reprezentálva a probléma az evolúciós keretrendszer számára.

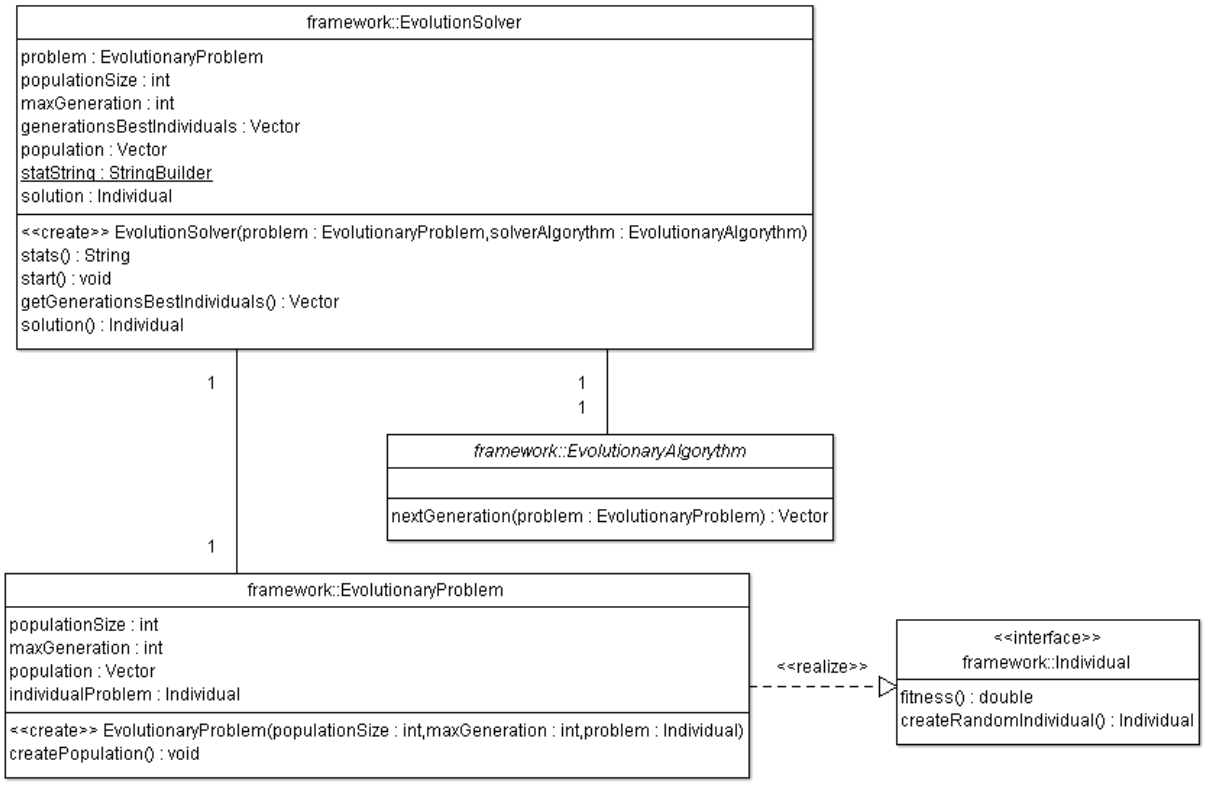
Az `EvolutionSolver` osztály feladata maga az evolúciós folyamat vezérlése. Konstruktora egyrészt a problémát reprezentáló, a már inicializált kezdeti populációt tartalmazó `EvolutionaryProblem` példányt vár, másrészt pedig egy olyan osztályt, mely implementálja az `EvolutionaryAlgorithm` interfészt. Ezekkel már minden adatunk

megvan, hogy elindítsuk a problémamegoldást. Az osztály implementálja a `Solution` interfészt, ami gondoskodik arról, hogy a megoldó algoritmus által visszaadott megoldást el tudjuk érni. Az evolúciós folyamat levezénylésének a magja a következőképpen néz ki:

```
int generation=0;
Vector<Individual> newGeneration;
do{
     newGeneration=new Vector<Individual>();
     //Az új egyedek előállatása
     newGeneration.addAll(
          solverAlgorithm.nextGeneration(problem));
     newGeneration.addAll(problem.getPopulation());
     //A legjobb egyedek kiválogatása miatt sorba kell rendezni
     //azokat
     Collections.sort(newGeneration);
     Vector<Individual> tmp=new Vector<Individual>();
     population.clear();
     int i=0;
     for (i=0;i<populationSize;i++) {
          Individual ind=(Individual)newGeneration.get(i);
          tmp.add(ind);
     }
     population.addAll(tmp);
     problem.setPopulation(population);
     generationsBestIndividuals.add(
          (Individual)population.get(0));
     generation++;
}while (generation<maxGeneration);
```

A ciklus magja pontosan annyiszor fut le, ahány generációt elő kell állítani. A `newGeneration` vektorba először belekerülnek a problémához tartozó aktuális populációból a keresztezési eljárással előállított egyedek. Ezt követően feltöltésre kerülnek az eredeti populáció elemei, így a vektor tartalmazza a következő generáció lehetséges egyedeit. A szelekciós eljárás egy `for` ciklusból áll, feladata hogy kiválogassa azon legjobb egyedeket, amelyek még belefférnek a következő populációba, a keretrendszeremben ez kizárólag fix méretű lehet.

Összegzésképp a következő osztálydiagram segít megérteni a keretrendszer struktúráját. A technikai metódusokat természetesen nem tartalmazza, csak amelyek az osztályok kommunikációjához szükségesek:



4.3. A keretrendszer használata

Kétféle módon lehet használni a keretrendszert a problémától függően. A konzolos forma az alap használati mód, bármely probléma esetén a paraméterek átadása a konzolról történik. Az utazó ügynök problémára grafikus felhasználói felület is írtam, így sokkal szemléletesebben lehet összehasonlítani a hozzá implementált algoritmusokat. Ezek lehetnek evolúciósak és általános jellegűek, ugyanis a városok kirajzolásához csupán a megoldás egy példányát kell visszaadni, azt pedig a `Solver` interfész implementálásával biztosítani lehet. A programot parancssorból a `java StartProgram` utasítással lehet indítani, melynek paraméterei a következők:

```
[-npop populationSize -ngen maxGeneration]
{-alg algorithmClassName}
{-p ProblemClassName [problemParameters]}
```

A `-alg` és `-p` kapcsolók használata kötelező. Az előbbivel a probléma-megoldó algoritmust, utóbbival pedig a problémát reprezentáló osztályt lehet megadni. A problémának lehetnek saját paraméterei, ezt közvetlenül az osztály után lehet átadni. Ha evolúciós

módszerrel akarjuk megoldatni a problémát, akkor a megoldó algoritmusnak implementálnia kell az `EvolutionaryAlgorithm` interfészt, valamint meg kell adni a következő paramétereket: a `-npop` paraméter esetében a populáció méretét lehet beállítani, a `-ngen` paraméterrel pedig az előállítani kívánt generációk számát lehet megadni. Ilyen esetben mindkét paraméter megadása kötelező, különben a program hibaüzenettel leáll és kiírja, hogy mely paraméterek hiányoznak. Bizonyos problémák esetében specifikus paraméterekre is lehet szükség, amelyeket a `ProblemParameters` helyére kell írni. Ha nem evolúciós módszerrel akarjuk megoldatni a problémát, akkor a `-alg` és `-p` kapcsolókra természetesen nincs szükség. Az ilyen osztályoknak a `NonEvolutionary` interfészt kell implementálniuk.

5. Az utazó ügynök probléma

A történelemben a gazdaság fejlődésének egyik alapvető mozgatórugója a termelés és a beruházások mellett – a kornak megfelelően – a különböző patriarchák, törzsek, városok stb. közötti kereskedelem volt. Ahogyan az áruszállítás egyre hatékonyabbá és gyorsabbá vált, a kereskedők előtt újabb lehetőségek nyíltak meg addig ismeretlen piacok bevonására. Ennek megfelelően a kereskedelmi útvonalak kiszélesedésével azok egyre bonyolultabbak lettek, olyannyira, hogy a lehető legrövidebb úton bejárni valamennyi kereskedelmi pontot egyre nehezebb feladattá vált.

Ezt a problémát ma utazó ügynök problémának hívják (röviden TSP), általános megfogalmazása a következő: adott városok egy halmaza és az azok között fennálló költségek. A cél az, hogy egy kezdeti városból kiindulva, majd a végén ugyanoda visszatérve minden várost pontosan egyszer érintsünk úgy, hogy ezt a lehető legkisebb költség mellett érjük el, ez lesz az optimális megoldás. Az egyszerűség kedvéért két város költségét a közöttük lévő euklideszi távolsággal jellemzem. A sorrendet, amely szerint bejárjuk a városokat **körút**nak nevezzük. A probléma nehézsége abból adódik, hogy ha *n* várost veszünk alapul, akkor a lehetséges körutak száma ((n-1)!)/2. Ebből igazán látszik, hogy viszonylag kevés pont esetén is nagy számítási kapacitásra van szükség az optimális megoldás előállításához. De a cél a való életben nem is a legrövidebb körút megtalálása, hanem annak egy elég jó megközelítését is elfogadhatjuk.

5.1. Optimalitási becslések

Léteznek módszerek arra, melyek a városok gráfjának ismeretében jó közelítést adnak a lehetséges körutak alsó korlátjáról. A legnagyobb hasznuk az, hogy a kereső algoritmusok eredményeit összehasonlítva az alsó becsléssel ítéletet mondhassunk azok hatékonyságáról. Két módszert mutatok be, az egyik az *1-fa*, másik pedig ennek a továbbfejlesztése, a Held-Karp korlát.

5.1.1. 1-fa korlát

Legyen adott egy *N* csúcsból álló gráf, jelöljünk ebben egy tetszőleges túrát *T*-vel. Ha *T*-ből eltávolítunk egy csúcsot – jelöljük ezt *v*-vel – akkor egy túra a következőkből fog állni:

egy feszítőfából a fennmaradt *N-1* ponton – jelöljük ezt *R*-rel – valamint a *v* ponthoz tartozó *e* és *e* élekből amelyekkel *R*-hez kapcsolódik. Az 1-fa korlát ennek a két komponensnek a minimumából áll elő: $t=l(R^*)$+költség(e^*)+költség(f^*), ahol R^* a minimális feszítőfa, az $l(R^*)$ ennek költsége, valamint e^* és f^* a *v* ponthoz kapcsolódó két legrövidebb él. Az 1-fa korlát általában 10%-kal az optimális útvonal értéke alatt van.

5.1.2. Held-Karp korlát

Az 1-fa korlát bizonyos esetekben elég pontatlan lehet, ugyanis előfordulhat, hogy a visszaadott élhalmaz távol esik az utazó ügynök túráktól, például nagyon nagy fokú pontok keletkezhetnek a feszítőfában, ezzel együtt sok egy fokú pont is létrejöhet. Erre egy megoldást a következő eljárás nyújt: válasszunk egy *v*-vel jelölt nagyfokú pontot a feszítőfában és a gráfban minden rá illeszkedő él súlyát növeljük meg *M*-mel. Jól megválasztott *M* esetén *v* kisebb fokú lesz a fában, azaz az 1-fa korlát értéke kevesebbel nő, mint *M* szorozva a *v*-hez kapcsolódó élek száma, de 2**M*-nél többel nőhet [2]. Ámde minden túra hossza pontosan 2**M*-mel nőtt, azaz jobb alsó korláthoz juthatunk.

Általánosan, hogyha *V* a csúcspontok halmaza, akkor az $u \in V$ csúcsponthoz hozzárendeljük a *k* csúcspont-értéket, akkor az *u*-ból kiinduló élek költségeit csökkentjük *k*-val. Tehát minden $u \in V$ $(u \neq v_0)$ csúcsponthoz hozzárendelhetjük az y_u csúcspont-értéket. Ezután minden $y = (y_u \mid u \neq v_0)$ vektorra meghatározzuk az R_y feszítőfát $V \setminus \{v_0\}$-ra. Ekkor Held-Karp alsó korlátnak nevezzük az

$$l(R_y) + 2 * \sum_{u \in V \setminus \{v0\}} y_u$$

értéket.

5.2. Körút statisztikák

Mahalanobis [1] szerint *n* elemű városhoz tartozó körutak optimális megoldásai nagyon közel állnak egymáshoz. Azaz, ha ugyanazon tartományból kijelölünk valamennyi várost, és ezt többször megismételjük, akkor a hozzájuk tartozó optimális körutak szórása viszonylag kicsi lesz. Az 5.1-es táblázatban 10 darab 1000 városból álló probléma optimális megoldásait láthatjuk, ahol a városok *x* és *y* koordinátáik a [0,1] tartományból egyenletes valószínűséggel lettek kijelölve.

5.1 táblázat 10 optimális körút 1000 város esetén

23.215	23.297	23.249	23.277	23.367
23.059	23.100	22.999	23.269	23.041

Mahalanobis megállapította, hogy n növekedésével a körutak hossza is $\sqrt{n}$-nel arányosan növekszik. Ghosh mérései alapján az optimális körút hossza nem lehet több mint $1.27\sqrt{n}$, Marks szerint pedig az elfogadható legkisebb érték $(\sqrt{n} - 1/\sqrt{n})/\sqrt{2}$. Beardwood, Halton és Hammersley [10] 1959-ben kiadott tanulmánya szerint, ha n tart a végtelenbe és az ezekhez tatozó optimális körutakat elosztjuk $\sqrt{n}$-el akkor azok egy β konstanshoz konvergálnak, ami körülbelül 0.7313.

Nagyon sok ország, sőt a világ összes városára is megoldották már az utazó ügynök problémát. A jelenlegi legjobb körút 7,515,877,991 méter hosszú, 1,904,711 várost tartalmaz, a becslések szerint 0.0487%-kal tér el az optimális megoldástól. Ezt Keld Helsgaun érte el idén májusban. Egy összefoglalót láthatunk a világ városainak körútjairól az 5.2 táblázatban [11].

5.2 táblázat A világ TSP útvonalainak története

Dátum	**Körút hossza**	**Heurisztikus algoritmus**
2001.10.31	7,539,742,312	Láncolt Lin-Karnighan
2001.11.18	7,532,949,688	Keld Helsgaun, LKH változat
2002.04.16	7,524,170,430	Keld Helsgaun, LKH változat
2003.03.18	7,518,528,361	Keld Helsgaun, LKH változat
2003.06.02	7,518,425,644	Nguyen et al.
2003.09.16	7,517,285,610	Keld Helsgaun, LKH változat
2004.07.26	7,516,122,185	Keld Helsgaun, LKH változat
2006.01.29	7,516,043,366	Keld Helsgaun, LKH változat
2008.11.07	7,515,947,511	Keld Helsgaun, LKH változat
2009.05.12	7,515,877,991	Keld Helsgaun, LKH változat

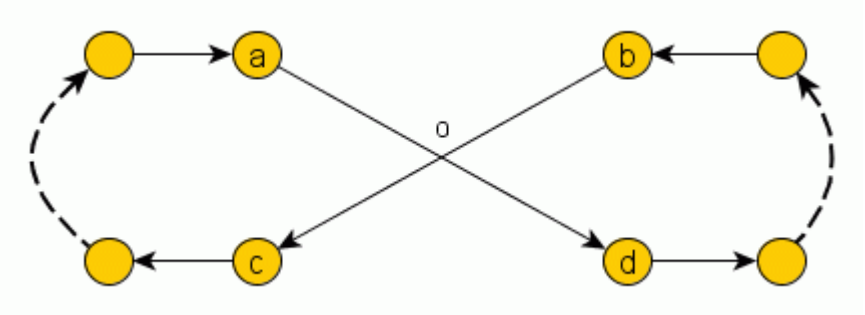
5.3. TSP- és más általános algoritmusok

5.3.1. Legközelebbi szomszéd

Az algoritmus úgy működik, hogy kiválasztunk egy tetszőleges várost, ez lesz a kiindulópont. Innen megmérjük a távolságot minden várostól, majd megnézzük, hogy melyik van legközelebb. A két csúcspont között élt húzunk, ez már biztosan része lesz a körútnak. Az újonnan beillesztett várost fogjuk most vizsgálni. Ennek is megmérjük a többitől való távolságát, kivéve azokat, melyek a körútban már szerepelnek, majd a legközelebbihez élt húzunk. Ezt addig folytatjuk, amíg már nem maradt lekötetlen város, az utolsót pedig összekötjük a kiindulóponttal.

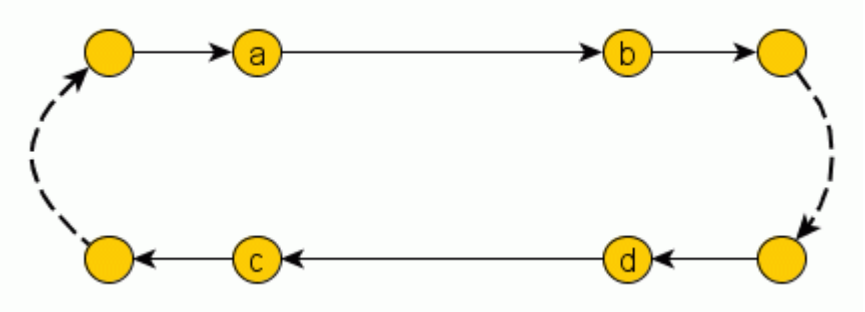
5.3.2. Kereszteződések eltávolítása

Ha a városok koordinátái adottak, akkor a megoldáskeresés a legtöbb esetben azok véletlenszerű összekötésével kezdődik, természetesen betartva azt a szabályt, hogy minden városból pontosan két irányba indul ki él. Sokféle technika létezik arra, hogy a körút hosszát, ha nem is a minimális mértékre, de lerövidítsük. Ha felrajzoljuk a koordinátákat és az azok közötti véletlen éleket, akkor szinte biztosan lesznek olyan részek, melyek átfedést tartalmaznak. Ezek ellen a következő képpen lehet védekezni:



5.1 ábra Kereszteződés a körútban

Látható, hogy az `a` és `d` valamint a b és c jelű városok közötti élek keresztezik egymást. Hogy lerövidítsük a körutat, az `(a,d)` és `(b,c)` éleket el kell távolítani, és e helyett az `(a,b)` és `(d,c)` éleket kell létrehozni.



5.2 ábra A körút a keresztezés megszüntetése után

Ezzel garantáltan lecsökken az úthossz, hiszen ha az `(a,d)` és `(b,c)` élek metszéspontját `o`-val jelöljük, akkor a háromszög egyenlőtlenség miatt a korábbi `[(a,o)+(o,b)]` szakasz hosszabb mint az `(a,b)` él, valamint `[(c,o)+(o,d)]` is hosszabb mint a `(c,d)` él [1]. Nem szabad figyelmen kívül hagyni az eljárás mellékhatását, hogy a `b` és a `d` csúcsok közötti körútrész iránya megfordult. Ez persze a céljaink elérését jelen esetben nem befolyásolja. Csak akkor lehet gond, hogy ha a költségmátrix nem szimmetrikus, azaz ha az `(u,v)` és a `(v,u)` élek költsége nem egyenlő (aszimmetrikus TSP). Ekkor az operáció helyétől függően a körút hossza jelentősen megváltozhat, de az ilyen problémák megoldásához nem is az euklideszi utazó ügynök problémára megalkotott algoritmusokat kell használni.

Ha az összes ilyen kereszteződést megszüntetjük, akkor az optimális értéktől csak pár százalékkal fog eltérni az így keletkezett körút hossza. Ez a technika alapjául szolgál sok hatékony útkereső algoritmusnak.

5.3.3. Beillesztési heurisztikák

Sok TSP algoritmus kiindulópontja nem az összes várost tartalmazó körút, hanem a városok egy részhalmazából megalkotott rész-körút. Az algoritmus valamilyen módszer szerint kiválasztja a még le nem fedett csúcspontokat és valamilyen heurisztika szerint beilleszti azt a körútba.

Már a kiinduló-körutak kijelölését is többféleképpen választhatjuk meg, mindegyik néhány darab csúcsponttal vagy egy éllel kezdődik. Ezek kijelölése történhet véletlenszerűen, vagy szisztematikusan, például a „konvex burok” technikával: határozzuk meg a csúcsok konvex burkát, és képezzünk belőle rész-körutat. Mivel ezek a városhalmaz peremén helyezkednek el, így már az eljárás elején kirajzolódik a körút kontúrja, persze ez a

későbbiekben módosulni fog. A kiinduló-körút alapján az eljárások többféle heurisztika szerint illesztik be a még le nem kötött csúcspontokat:

- Megkeresik azt a pontot, melynek a távolsága minimális lesz a rész-körúttól, majd oda szúrják be, ahol a körút hosszában ez a legkisebb növekedést idézi elő. A kiinduló-körút a legkisebb költségű él a gráfban.
- Megkeresik a körúttól a legtávolabbi pontot, majd oda szúrják be, ahol a legkisebb lesz a hossznövekedés. A leghosszabb él lesz a kiinduló-körút.
- Azt a pontot választják ki, melynek a beillesztésével a hossznövekedés minimális lesz. A kiinduló-körútra nincs javaslat.

5.3.4. Christofides heurisztika

Ez az algoritmus mindmáig a legjobb elméleti korlátot adja az utazó ügynök problémára. Az algoritmussal előállított körút hossza az optimális legfeljebb 3/2-ed szerese, működése a következő [2]:

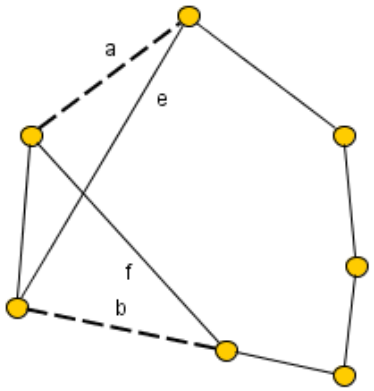
- Állítsuk elő a gráfban a minimális súlyú feszítőfát, jelöljük *F*-fel.
- A feszítőfában keressük meg azokat a csúcsokat, melyek páratlan fokúak, jelöljük ezek halmazát *W*-vel.
- Keressük meg a *W* által feszített minimális súlyozott részgráf minimális súlyú teljes párosítását, jelöljük ezt *M*-mel.

Tekintsük az F∪M gráfot, ami a mindkettőben szereplő éleket két példányban tartalmazza. Ennek a gráfnak minden foka páros, tehát van Euler-séta. (Euler-séta alatt egy gráf olyan zárt élsorozatát értjük, mely a gráf összes élét pontosan egyszer tartalmazza. Megengedett, hogy egy ilyen séta egy ponton többször is áthaladhat.) Ha minden csúcs foka 2, akkor egy túrát kaptunk, tehát leállunk. Ha van legalább 4 fokú csúcs, akkor az Euler sétának erre illeszkedő egymás utáni két élét egy éllel helyettesítjük. Az így kapott gráfban továbbra is van Euler-séta. Ilyen műveleteket csinálunk egészen addig, amíg túrát nem kapunk. Az algoritmus csak a háromszög-egyenlőtlenséget kielégítő nemnegatív súlyok esetén működik.

5.3.5. A Lin-Kernighan módszer

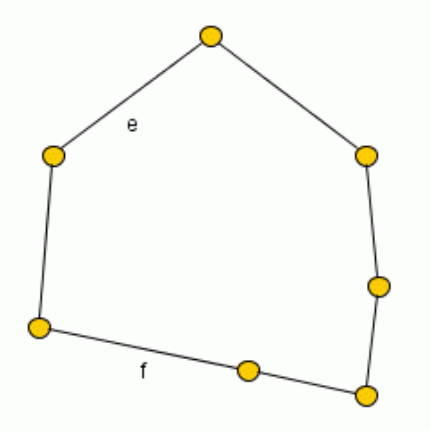
1973-ban alkotta meg Lin és Kernighan a nyolcvanas évek végéig a leghatékonyabbnak tartott algoritmust. Többféle változata létezik, de mindegyiknek ugyan az az alapja. Máig is ez a legkedveltebb módszer a nagyméretű utazó ügynök problémák megoldására, a világ városainak eddig ismert legrövidebb körútját is ezzel a módszerrel határozták meg. Az eljárás, amit ismertetni fogok nem az eredeti algoritmus, annak csak a váza.

A Lin-Kernighan algoritmus egy útvonal javító eljárás. A kiindulópontja egy tetszőlegesen előállított körút, ezen végez olyan műveleteket, mellyel drasztikusan lecsökkentheti az útvonal hosszát. A legfontosabb fogalma a *k-opt* művelet: az eredeti túrából törlésre kiválasztunk *k* élt, majd az így keletkezett részeket úgy próbáljuk meg összekötni, hogy az eredeti túrától jobb körutat kapjunk. Tulajdonképpen itt az egyes szegmensek és az azoknak vett inverzének egy permutációjából állítjuk elő az új útvonalat. Ezt a lépést annyiszor végezzük el, amennyiszer csak tudjuk. Ha ezzel a módszerrel már nem tudunk javítani a körút hosszán, akkor mondjuk azt, hogy az útvonal *k-optimális*. A *k* növekedésével egyre növekszik az eljárás időigénye, az implementálás nehézségéről nem is beszélve. Speciális esetei a gyakran használatos *2-opt* és *3-opt* műveletek. Nézzünk egy példát egy *2-opt* műveletre: a következő ábrán látható egy 7 városból álló körút, melyről ránézésre megállapíthatjuk, hogy nem optimális, ugyanis vannak benne olyan – az alábbi ábrán *e*-vel és *f*-fel jelölt – élek, melyek keresztezik egymást:



Egyetlen *2-opt* lépéssel el tudjuk tüntetni a kereszteződés a következő módon: válasszunk ki két élt törlésre, ez esetben célszerű az *e* és *f* élt, ezzel kapunk két utat.

Keressünk másik két élt, mely még nem szerepelt a körútban és velük szabályos körút keletkezik úgy, hogy a hossza kisebb legyen az eredetinél. Erre tökéletesen alkalmasak az *a*-val és *b*-vel jelölt élek, a művelet elvégzése után a körút következőképpen néz ki:



Szemmel láthatóan a keletkezett körút jobb, mint az eredeti, ez persze a nagyobb problémák esetén egyetlen *2-opt* lépéssel nem jönne létre, mindenesetre a kereszteződéseket sikerült eltüntetni.

Lin és Karnighan módszere nem fix *k* értékkel dolgozik, hanem időben változik. A cél az, hogy addig növelje ezt az értéket, míg *k*-t nem maximalizálta. Minden iteráció során újra meghatározásra kerül az értéke, mely a következőképpen zajlik: amikor az algoritmus *n-opt* lépésre készül, végrehajt egy tesztsorozatot, melyből megállapítja, hogy az *(n+1)-opt* műveletet érdemes-e végrehajtani. Ezt a lépést addig fogja folytatni, amíg egy megadott végfeltétel be nem következik.

5.3.5.1. A Lin-Kernighan algoritmus

Az algoritmus váza a következő [3]:

---

1. Adott egy T egy véletlen kezdő útvonal.
2. i:=1, válasszuk ki t_1-et
3. Válasszuk ki x_1-et, ahol $x_1 = (t_1, t_2) \in T$
4. Válasszuk ki y_1-et, ahol $y_1 = (t_2, t_3) \notin T$ és $G_1 > 0$
   Ha ez nem lehetséges, ugorjunk a 12. lépésre
5. i:=i+1
6. Válasszuk ki x_i-t, ahol $x_i = (t_{2i-1}, t_{2i})$ és teljesülnek:

a. ha a t_{2i}-t csatlakoztatjuk t_1-hez, akkor az így keletkező T' egy túra

b. $x_i \neq y_s$ minden $s<i$-re teljesül

Ha T' jobb túra mint T, akkor T:=T' és ugorjunk a 2. lépésre

7. Válasszuk ki y_i-t, ahol $y_i = (t_{2i}; t_{2i+1})$ és teljesülnek:
   a. $G_i>0$
   b. $y_i \neq x_s$ minden $s \leq i$-re teljesül
   c. x_{i+1} létezik

   Ha y_i létezik, akkor ugorjunk az 5. lépésre
8. Ha van még ki nem próbált alternatíva y_2-re, akkor i:=2 és ugorjunk a 7. lépésre
9. Ha van még ki nem próbált alternatíva x_2-re, akkor i:=2 és ugorjunk a 6. lépésre
10. Ha van még ki nem próbált alternatíva y_1-re, akkor i:=1 és ugorjunk a 4. lépésre
11. Ha van még ki nem próbált alternatíva x_1-re, akkor i:=1 és ugorjunk a 3. lépésre
12. Ha van még ki nem próbált alternatíva t_1-re, akkor ugorjunk a 6. lépésre
13. Állj (vagy ugorjunk az 1. lépésre)

5.3.6. Korlátozás és szétválasztás módszere

A korlátozás és szétválasztás módszerének alapötlete az, hogy a feladatot több részfeladatra bontjuk szét. Az eredeti feladat megoldáshalmaza a részfeladatok megoldáshalmazainak diszjunkt uniójából áll. A részfeladatok további szétbontásával fa keletkezik, melynek a gyökerében található az eredeti feladat. Az eredeti feladat optimum értéke megegyezik a részfeladatok optimumértékeinek maximumával. A *korlátozás* azt jelenti, hogy minden részfeladatnál felső korlátot számolunk az optimum értékére. Amennyiben egy részfeladatra vonatkozó felső korlát kisebb, mint egy már ismert alsó korlát, akkor azzal a részfeladattal nem kell tovább foglalkozni, törölhetjük a fából.

A korlátozás és szétválasztás algoritmusa a következő [4]:

1. **Kezdeti körút:** határozzunk meg egy kezdeti körutat, jelöljük z^*-gal a hozzá rendelt célfüggvény-értéket (ez lesz a felső korlát)

2. **Kezdeti relaxáció:** oldjuk meg azt az úgynevezett relaxált problémát, melyet úgy kapunk, hogy elhagyjuk a problémából a részutakat kizáró megszorításokat, majd jelöljük $\bar{z}_1$-el egy hozzá rendelt célfüggvény értéket.
    a. Ha $z_1 > z^*$, akkor állj, a heurisztikus megoldás optimális.
    b. Különben P_1 jelölje a fában az eredeti problémát és lássuk el az „élő csúcs" megjegyzéssel, mely azt jelzi, hogy a csúcs tartalmaz olyan rész-körutat, melynek nem ismerjük az optimális megoldását.
3. **Korlátozás és szétválasztás eljárás:** nézzük meg, hogy van-e még élő csúcs a fában.
    a. Ha igen: Válasszunk egy P_k élő csúcsot legjobb alsó korláttal, majd ugorjunk a 4. lépésre.
    b. Ha nem: A fában az ismert legjobb túra optimális.
4. Megnézzük, hogy P_k tartalmaz-e rész-utakat.
    a. Ha igen: Ugorjunk az 5. lépésre
    b. Ha nem: Állj, a P_k csúcsban lévő megoldás optimális.
5. **Korlátozás:** Jelöljük ki azt a rész-körutat, mely azon élekből áll, amelyet az 1. lépésben a kezdeti körút megalkotásánál nem fixáltunk le, jelöljük ezt a következő módon: $\{1,2,\ldots,r,1\}$. Vágjuk a P_k-t csúcsot P_{s+1} P_{s+2} .. P_{s+r} részekre úgy, hogy P_{s+1} csúcsban legyen $x_{1,2}:=0$, és minden P_{s+j} $(j=2,\ldots,r)$ csúcsban $x_{j,j+1} := 0$ (ahol $x_{r,r+1}=x_{r,1}$) és $x_{i,i+1} := 1$ $(i=1,\ldots,j)$
6. **Vágás:** Minden P_{s+j}-re csináljuk meg a következőket:
    a. Oldjuk meg a benne levő rész-problémát az 5. lépésben meghatározott változókkal.
    b. Az ehhez tartozó célfüggvény-értéket jelöljük $\bar{z}_{s+j}$-vel.
    c. Ha $z_{s+j} \geq z^*$, akkor hagyjuk el P_{s+j}-t.
    d. Ha $z_{s+j} < z^*$ és a megoldás nem tartalmaz rész-utakat, akkor $z^* = \bar{z}_{s+j}$.
    e. Ha $z_{s+j} < z^*$ és a megoldás tartalmaz rész-utakat, akkor P_{s+j} élő csúcs.

    Ugorjunk a 3. lépésre.
7. **Vége:** az algoritmus az optimális körúttal tér vissza.

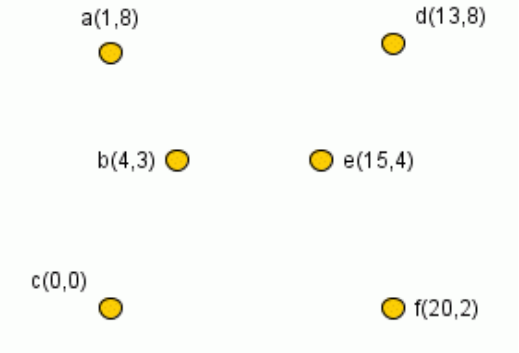
5.3.7. Kruskal-féle mohó algoritmus

A leghatékonyabb mohó algoritmus, amit bemutatok Kruskal ötletén alapul, ami részben optimális körutat állít elő és csakis teljes gráfon működik. Működése a következő:

1. Rendezzük az éleket a súlyuk szerinti növekvő sorrendbe!
2. Válasszuk ki a legrövidebb élt. Tekintsük sorban az éleket és válasszunk ki egyet közülük egy már korábban kiválasztott éllel együtt a következő megszorításokkal:
    a) Egy pontnak maximum 2 lehet a fokszáma.
    b) Nem jöhet létre kör, hacsak a kiválasztott élek száma nem egyezik meg a gráfban található pontok számával. Ekkor a körút teljes, ez lesz az algoritmus megoldása.

Nézzünk egy példát 6 várossal:

5.6 ábra



A rendezett élek halmaza sorrendben a következők:

((*d*, *e*), 4.4**)** , **(**(*b*, *c*), 5**)** , **(**(*a*, *b*), 5.1**)** , **(**(*e*, *f*), 5.3**)** , **(**(*a*, *c*), 8.1**)** , **(**(f, d), 9,2**)** , **…**

Az algoritmus a legrövidebb élek kiválasztásával kezdődik, majd ha valamelyik beleütközik a fent leírt megszorítások valamelyikébe, akkor törlésre kerül sor. Lépései a következők:

Kiválasztjuk a (d,e) élt.

Kiválasztjuk a (b,c) élt.

Kiválasztjuk az (a,b) élt.

Kiválasztjuk az (e,f) élt.

Töröljük az (a,c) élt, mert kört képez az (a,b) és (b,c) élekkel.

Töröljük az (f,d) élt, mert kört képez a (d,e) és (e,f) élekkel.

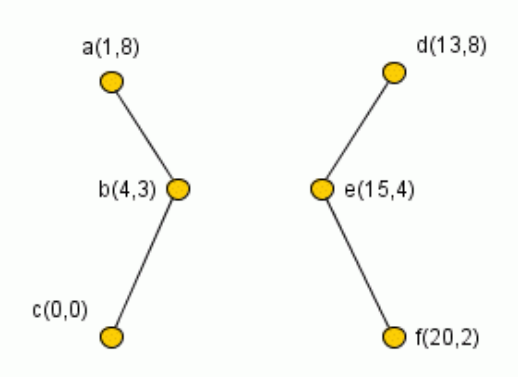
⋮

Kiválasztjuk a (c,d) élt.

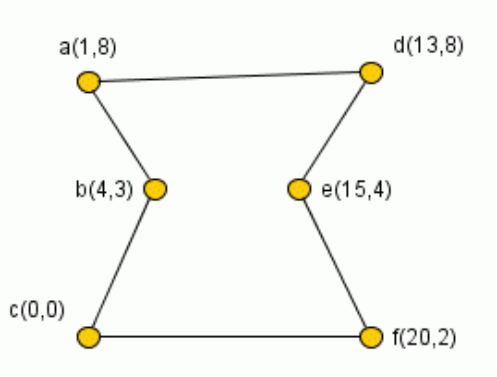
Kiválasztjuk az (a,f) élt, ebben az esetben kör jön létre, de mivel már az összes élt végignéztük, így ez adja a megoldást.

Az így keletkezett körút hossza 65.1, ami körülbelül 5%-kal tér el az optimális megoldástól. Az 5.7.a ábra az első négy lépés megtétele utáni keletkezett közbülső állapotot szemlélteti, az 5.8.b ábra pedig az algoritmus által visszaadott megoldást ábrázolja.

5.7.a ábra



5.7.b ábra



5.3.8. Tabu keresés

A tabu keresés egy olyan metaheurisztukus algoritmus, amely kombinatorikus optimalizációs problémákra nyújt hatékony megoldást. Nagyon sok megoldáskereső algoritmusnak megvan az a hibája, hogy a keresés egy lokális optimum körül ingadozik, ezt a hibát küszöböli ki az eljárás. Alapvetően egy olyan iterációs eljárás, mely emlékezettel rendelkezik, és a folyamat középpontjában pedig lokális vagy szomszédsági kereső algoritmus áll. A memóriáját arra használja, hogy a többi megoldáskereső algoritmustól eltérően nem csak az eddigi legjobb megoldást tárolja el, hanem azokat is, melyeket az algoritmus a keresés során már megvizsgált. Ez az úgynevezett *tabu lista*. Ha a keresés során újra előállítja a lista egy elemét, akkor azt tabunak nyilvánítja, és más irányban próbálkozik. Ez segít abban, hogy a lokális optimum körüli térből kiszakadjon és más szélsőértékek felé induljon el. Ha jól használjuk, egy akkor optimum közeli megoldást garantálni tud.

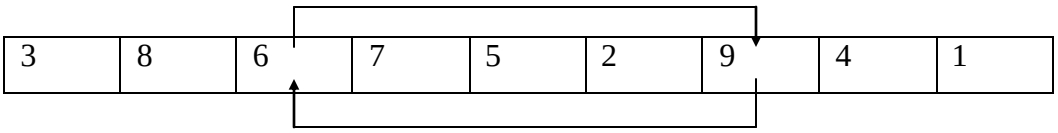
Az utazó ügynök problémára az algoritmus általánosan a következő [5]:

1. **Megoldás reprezentáció:** a lehetséges megoldások csúcsok sorozatával vannak reprezentálva, minden csúcs pontosan egyszer szerepel, sorrendjük pedig a körútban betöltött helyük szerint van meghatározva:

Egy megoldás reprezentációja

3	8	9	2	5	7	6	4	1

2. **Kezdeti megoldás:** egy még nem optimális megoldás leggyorsabb előállítására a mohó algoritmus javasolt.
3. **Szomszédság:** egy megoldás szomszédsága alatt olyan más megoldásokat értünk, melyeket úgy nyerünk, hogy az eredeti megoldásban két csúcs között felcseréljük a sorrendet. Ezzel a művelettel nem sértjük meg az utazó ügynök problémára tett megszorításokat, azaz nem keletkeznek új csúcspontok, de nem is vesznek el, csak a sorrendjük változik meg:



Két elem felcserélésével előálló megoldás

4. **Tabu lista:** annak érdekében, hogy a folyamat műveleti területe ne korlátozódjon egy szűk megoldáshalmazra – azaz ne jöjjön létre ciklikusság – a tabu listában eltároljuk azokat a csúcspárokat – a fenti esetben például a (6,9)-et – melyeket már alkalmaztunk a kiinduló megoldásra. Amikor az algoritmus a kiinduló megoldás újabb szomszédját akarja előállítani, előtte megbizonyosodik afelől, hogy a végrehajtáshoz kiválasztott csúcspár nem szerepel a listában. A lista természetesen véges, ha megtelt, akkor a legrégebbi elemeit elhagyja és az újonnan érkezőket teszi a helyére. Ez nagyon hasznos lehet, ugyanis ha jobb megoldásjelöltet talált az algoritmus, akkor annak más szomszédsága lesz, így más tabu lista alkalmas hozzá.
5. **Elvárások a megszorítássokkal szemben:** a tabu lista bizonyos esetekben túl szigorú megszorítást definiál: megakadályozhat olyan lépéseket is, amikor a

ciklikusság veszélye nem áll fent, vagy a megoldástér egy részhalmazán túl meggátolja a keresést. Emiatt szükség lehet arra, hogy a tabu listából elemeket távolítsunk el. A kritériumot arra használjuk, hogy olyan lépést is megtehessünk, ami tabu, de ezzel olyan megoldást találunk, mely a jelenleg ismert legjobb megoldásnál is jobb tulajdonsággal rendelkezik.

6. **Változtatások:** a keresés elég gyakran megreked egy lokális optimum körül. Annak érdekében, hogy más területekre is kiterjesszük a keresést – a globális optimum feltárásához – változtatásokat kell eszközölni az eljárásban. A *gyakoriságon alapuló memória* eltárolja az egyes műveletek gyakoriságát, azaz minden lépéshez eltárolja, hogy hányszor lett alkalmazva a keresés során. Ha valamely lépésről kiderül, hogy nem sikerült vele javítani az eddigi megoldáson, akkor az nagyobb gyakorisági értéket kap, így kisebb a valószínűsége hogy a jövőben alkalmazásra kerül. Ha egy előre meghatározott iterációs lépés megtétele után sem talált jobb megoldást, akkor az alacsonyabb gyakoriságú lépéseket veszi előre az algoritmus, így nagyobb területen folytatódhat a keresés. Egy 10 városból álló problémához tartozó memória táblázat a következő képpen nézhet ki a keresés során:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1		4				2			
2				5					
3								5	5
4		2			1				
5								4	
6	8		3						
7						6			
8	1					2			3
9				7					

A táblázatban az *i*-edik sor *j*-edik eleme azt mutatja, hogy az (*i*,*j*) csúcs hányszor lett kiválasztva.

7. **Terminálási feltétel:** az algoritmus befejezi a működését, ha az iterációk száma eléri az előre meghatározott értéket.

5.3.9. Szimulált hűtés

A kohászatban a fémek megerősítéséhez úgynevezett temperálási eljárást alkalmaznak, ami során az anyag kristályrácsszerkezete jól strukturált lesz és a belső energiája minimálisra csökken. Ezt úgy érik el, hogy az anyagot olvadáspontra felmelegítik, a fém atomjai így szabadon elmozdulnak, majd a lassú *hűtést* követően az atomok felvesznek egy optimálisabb szerkezetet. Ezt addig alkalmazzák, míg az anyag el nem éri a kívánt állapotot.

5.3.9.1. Iterációs hegymászó algoritmus

Az algoritmus tárgyalása előtt nézzük meg, hogyan működik az iterációs hegymászó algoritmus, mert a szimulált hűtés nem sokban különbözik tőle [6]:

1. t:=0, *best*:=0
2. **repeat**
3. *local*:=FALSE
4. válasszuk ki $\mathbf{v}_c$-t véletlenszerűen
5. **repeat**
6. válasszuk ki az összes új pontot $\mathbf{v}_c$ szomszédságában
7. válasszuk ki $\mathbf{v}_n$-t az új pontok halmazából úgy, hogy az *eval* kiértékelő függvény $\mathbf{v}_n$-re a legjobb értéket adja
8. **if** eval($\mathbf{v}_n$) jobb mint eval($\mathbf{v}_c$)
9. **then** $\mathbf{v}_c$:=$\mathbf{v}_n$
10. **else** *local*:=TRUE
11. **until** *local*
12. t:=t+1
13. **if** $\mathbf{v}_c$ értéke jobb, mint *best* **then**
14. *best*:=$\mathbf{v}_c$
15. **until** t=MAX
16. **end**

A belső ciklus mindig a lokális optimumot adja vissza. A külső ciklusban mindig az egész megoldásteret figyelembe véve generál egy teljesen új pontot és ennek a környezetében

keresi meg a legalkalmasabb megoldást. A lokális optimum körüli beragadás ellen így jó megoldást nyújt. Összesen MAX iterációs lépés után visszaad egy megoldást a *best* változóból.

Módosítsuk az eljárást a következő változtatásokkal:

- Ahelyett, hogy $\mathbf{v}_c$ környezetében az összes pontot vizsgálnánk és keresnénk meg a legjobbat közülük, csupán egyet válasszunk ki, $\mathbf{v}_n$-t!
- Válasszuk ki ezt az új pontot egy bizonyos valószínűséggel mely a $eval(\mathbf{v}_n)$ és $eval(\mathbf{v}_c)$ különbségén alapszik!

Az így nyert algoritmust sztochasztikus hegymászó algoritmusnak hívjuk.

5.3.9.2. Sztochasztikus hegymászó algoritmus

Pszeudó kódja a következő[6]:

1. t:=0
2. válasszunk ki egy véletlenszerű $\mathbf{v}_c$ pontot
3. **repeat**
4. válasszunk ki egy $\mathbf{v}_n$ pontot $\mathbf{v}_c$ környezetéből
5. fogadjuk el $\mathbf{v}_n$-t $\frac{1}{1+e^{\frac{eval(v_n)-eval(v_c)}{T}}}$ valószínűséggel
6. t:=t+1
7. **until** t=MAX
8. **end**

Az eljárásban egyetlen ciklus szerepel, valamint az új pont elfogadása már nem evidens, hanem egy p valószínűségtől függ. Így előfordulhat, hogy az új elfogadott pont rosszabb lesz, mint az aktuális. A p pont függ egyrészt az $eval(\mathbf{v}_n)$-$eval(\mathbf{v}_c)$ értéktől, másrészt pedig egy T konstanstól. Ezen T érték növekedésével a p valószínűsége $\frac{1}{2}$ felé tart, T=1 esetén pedig az algoritmus megegyezik a hagyományos hegymászó algoritmussal – ami a fent tárgyalt iterációs változatában a belső ciklussal egyezik meg. Kis T érték (például 10) esetén, ha az $eval(\mathbf{v}_n)$ és $eval(\mathbf{v}_c)$ értéke megegyezik, akkor p értéke körülbelül $\frac{1}{2}$, ha vizsgált pont sokkal jobb, mint a jelenlegi legjobb megoldás, akkor 1-hez közelít.

A szimulált hűtés algoritmusa abban különbözik a sztochasztikus hegymászó algoritmustól, hogy a T értéke nem konstans, hanem a futás során változik, a fizikai példával párhuzamban ez jelképezi a hőmérsékletet. Kezdetben ez magas – ekkor nagyon hasonlít egy egyszerű véletlenszerűen kereső algoritmushoz –, majd a futás során ez fokozatosan csökken, mindinkább hasonlítani fog egy hagyományos hegymászó algoritmushoz. A szimulált hűtés algoritmusa a következő [6]:

1. t:=0
2. inicializáljuk T-t
3. véletlenszerűen válasszuk ki a $\mathbf{v}_c$ pontot
4. **repeat**
5. **repeat**
6. $\mathbf{v}_c$ szomszédságában válasszunk ki egy $\mathbf{v}_n$ pontot
7. **if** *eval*($\mathbf{v}_c$) jobb mint *eval*($\mathbf{v}_n$) **then**
8. $\mathbf{v}_c$:=$\mathbf{v}_n$
9. **else if** *random*[0,1) < $e^{\frac{eval(v_c)-eval(v_n)}{T}}$ **then**
10. $\mathbf{v}_c$:=$\mathbf{v}_n$
11. **until** (terminálási feltétel)
12. *T:=g(T,t)*
13. t:=t+1
14. **until** (megállási feltétel)
15. **end**

A g(T,t) függvény állítja be a T új értékét, mint látható a megtett iterációs lépések számától valamint a jelenlegi T-től függ.

6. Az utazó ügynök probléma implementálása

6.1. Reprezentációs fajták

Az utazó ügynök problémát többféleképpen is lehet reprezentálni, de alapvetően két osztályba tudjuk sorolni a reprezentációkat:

- vektor alapú reprezentáció
- mátrix alapú reprezentáció.

Előbbinek nagyon sok változata létezik, de bármelyiket is választjuk, a számítógépes implementálás során nagyon meg kell gondolni, hogy melyiket választjuk, hiszen az optimális körút keresése során a probléma példánya folyton változáson megy keresztül, aminek reprezentációtól függően komoly költsége lehet. A városokat legtöbbször koordináta-rendszerben ábrázolják, minden várost két értékkel azonosítanak, az *x* és *y* koordinátáikkal. De nekünk nem is igazán a térbeli elhelyezkedésük a fontos, abban inkább a fitness függvénynek van érdekeltsége a körút kiszámításánál. A városokat általában 1-től *n*-ig terjedő egész számokkal reprezentáljuk, azaz megszámozzuk őket.

6.1.1. Mátrix alapú reprezentációk

Tekintsünk egy 9 városból álló körutat: (3 1 2 8 7 4 6 9 5), majd töltsünk fel egy 9x9-es mátrixot a következő módon: az *i*-edik sor *j*-edik oszlopába írjunk 1-et akkor és csak akkor, ha az *i*-edik város a körútban megelőzi a *j*-ediket, különben írjunk 0-t. Ezzel a következő mátrixot kapjuk:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	0	1	1	1	1	1	1
2	0	0	0	1	1	1	1	1	1
3	1	1	0	1	1	1	1	1	1
4	0	0	0	0	1	1	0	0	1
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	1
7	0	0	0	1	1	1	0	0	1
8	0	0	0	1	1	1	1	0	1
9	0	0	0	0	1	0	0	0	0

A mátrixban pontosan $\frac{n*(n-1)}{2}$ darab 1-es szerepel, a főátlóban kizárólag 0 szerepel, hiszen minden elem egyszer szerepel és egy elem sem előzheti meg saját magát a körútban. Továbbá a tranzitív tulajdonság is jelen van, azaz ha $m_{ij}=1$ és $m_{jk}=1$, akkor $m_{ik}=1$. Hiszen ha az *i*-edik elem megelőzi *j*-ediket és a *j*-edik megelőzi *k*-adikat, akkor az *i*-ediket is megelőzi a *k*-adik elem.

Másik megközelítése a mátrixos reprezentációnak Seniw ötletén alapul. Ő úgy töltötte fel a mátrixot, hogy az *i*-edik sor *j*-edik oszlopába csakis akkor írt 1-est, ha az *i*-edik elem közvetlen rákövetkezője a *j*-edik elem volt. Így egy olyan ritka mátrixot kapunk, amelyben minden sorban és oszlopban pontosan egy darab 1-es szerepel, a többi helyre 0 kerül. Az előző (3 1 2 8 7 4 6 9 5) példa Seniw-féle [7] reprezentációban a következőképpen néz ki:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0
4	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	1	0	0	0
9	1	0	0	0	0	0	0	0	0

Első ránézésre egyik módszernél sem könnyű megállapítani a városok sorrendjét, azaz egyik sem „emberbarát” reprezentáció, az igazi előnyük inkább a velük végzett műveletek során mutatkozik meg.

6.1.2. Vektor alapú reprezentációk

Kiindulásként a városokat a fent említett módon itt is megsorszámozzuk, majd különböző módszerek szerint vektorba rendezzük a körútban elhelyezkedő pozíciójuk alapján.

A **szomszédsági reprezentáció** esetében egy *n* elemű listát készítünk úgy, hogy a lista *i*-edik helyén pontosan akkor szerepel a *j* elem, ha a körútban az *i*-edik elem után közvetlenül a *j* elem következik. Egy példán könnyebb megérteni: tekintsük a (1,5,6,8,9,3,7,4,2) körutat, melynek a szomszédsági reprezentációja a következőképpen néz ki:

5 – 1 – 7 – 2 – 6 – 8 – 4 – 9 – 3

A legelső pozícióban az 5-ös szerepel, ugyanis az 1-es város után közvetlenül az 5-ös áll, a második pozícióban pedig 1-es szerepel, mert a 2-es város után közvetlenül az 1-es város szerepel.

A **sorrendi reprezentáció** is egy *n* elemű listával dolgozik, ahol az *i*-edik eleme egy [1, *n*-*i*+1] tartománybeli szám. A lista feltöltését egy példán keresztül mutatom be. Legyen a C=(1,2,3,4,5,6,7,8,9) vektor, mely sorrendben tartalmazza a városokat, valamint az előző reprezentációnál tekintett körút: (1,5,6,8,9,3,7,4,2). Vegyük a körút első elemét (1-es város), majd nézzük meg, hogy a C vektorban hányadikként szerepel az 1-es elem és bővítsük az *l* listát az 1-es elemmel, majd töröljük ki a C vektor első elemét. A körút következő eleme az 5-ös város, az átalakított C vektorban ez a negyedik helyen szerepel, így bővítsük ki az *l* listát a 4-es elemmel és töröljük a C vektor negyedik elemét. A körút harmadik eleme a 6-os, az új C vektorban a negyedik helyen szerepel, bővítsük a listát a 4-es elemmel a C-ből pedig töröljük a negyedik elemet. Az eljárás végén az *l* lista:

1 – 4 – 4 – 5 – 5 – 3 – 3 – 2 – 1

Az **út-reprezentáció** a legtermészetesebb az utazó ügynök probléma körútjának reprezentálására, már korábban is alkalmaztam, amikor a vektort abban a sorrendben töltöttem fel a városokkal, ahogyan a körútban helyezkednek el. A (1,5,6,8,9,3,7,4,2) körút az **út-reprezentációval** a következőképpen néz ki:

1 – 5 – 6 – 8 – 9 – 3 – 7 – 4 – 2

6.1.2.1. Keresztezési operátorok

Az út-reprezentációhoz három keresztezési eljárást mutatok be, melyeknek az a célja, hogy megváltoztassa a körutakat. Az eljárás után a keletkezett körút nem feltétlen lesz rövidebb.

PMX operátor: tekintsünk a keresztezni kívánt szülő-körutat, majd vegyünk mellé egy olyan körutat, mely a városokat sorrendben tartalmazza:

p_1=(1 2 3 4 5 6 7 8 9)

p_2=(4 2 8 7 3 1 9 5 6)

Véletlenszerűen jelöljünk ki mindkét szülőben ugyanazon a helyen két vágási pontot, majd a két gyermek körútba kerüljenek bele a két vágási pont közötti részutak úgy, hogy a p_1 szülő középső részútja kerüljön a gy_2 gyermekbe, a p_2 szülőé pedig a gy_1 gyermekbe, azaz cseréljük

fel a városok helyét. Az „*x*” jelöli a jelenleg ismeretlen várost, a | pedig a vágási pontokat:

gy_1=(x x | 8 7 3 1 | x x x)

gy_2=(x x | 3 4 5 6 | x x x)

A következő cserék történtek: $3 \leftrightarrow 8, 4 \leftrightarrow 7, 5 \leftrightarrow 3, 6 \leftrightarrow 1$

Az „*x*”-ek helyére sorrendben írjuk be a városokat az eredeti szülőjükből addig, amíg nem jön létre duplikált város, az ilyeneket egyszerűen hagyjuk ki:

gy_1=(x 2 | 8 7 3 1 | x x 9)

gy_2=(x 2 | 3 4 5 6 | 9 x x)

A megmaradt üres pozíciókba a fent említett cserék alapján kerüljenek a városok, például a gy_1 első pozíciójába a p_1-ből az 1-es kerülne, de helyette a $6 \leftrightarrow 1$ csere alapján 1-es kerül. A két gyermek a következő lesz:

gy_1=(6 2 | 8 7 3 1 | 4 3 9)

gy_2=(7 2 | 3 4 5 6 | 9 8 1)

OX operátor: az előző példa alapján vegyük p_1 és p_2 szülőket, majd két vágási pontot létrehozva tartsuk meg a középső rész-utakat:

gy_1=(x x | 3 4 5 6 | x x x)

gy_2=(x x | 8 7 3 1 | x x x)

A második vágási ponttól kezdve folyamatosan, ugyanabban a sorrendben másoljuk a városokat a gyermekekbe, a második szülő ekkor a következő képpen fog kinézni:

9 – 5 – 6 – 4 – 2 – 8 – 7 – 3 – 1

Töröljük ebből a másik (az első) szülő középső részútjának elemeit, azaz a 4, 5, 6 és 7-es elemeket, így a következő marad:

9 – 2 – 8 – 7 – 1

Ezek az elemek végül sorrendben bekerülnek az első gyermekbe a második vágási ponttól kezdődően:

gy_1=(8 7 1 3 4 5 6 9 2)

Hasonlóan kapjuk a második gyermeket is:

gy_2=(5 6 8 7 3 1 9 2 4)

CX operátor: tekintsük a fenti p_1 és p_2 szülőket. A gyermek kezdetben teljesen üres, majd az első szülő legelső elemét, az 1-est beteszi az első helyre, majd a következőket csinálja: az első szülő elemein folyamatosan végigmegy úgy, hogy megnézi, hogy a második

szülőben a vizsgált elem alatt mely város szerepel. Ha azt be tudja tenni a gyermekbe anélkül, hogy duplikáció lépne fel, akkor beteszi, ha nem, akkor onnantól kezdve a gyermek üres pozícióiba a második város sorrendben megfelelő elemét teszi bele. A gyermek a duplikált város megjelenése előtt a következőképpen néz ki:

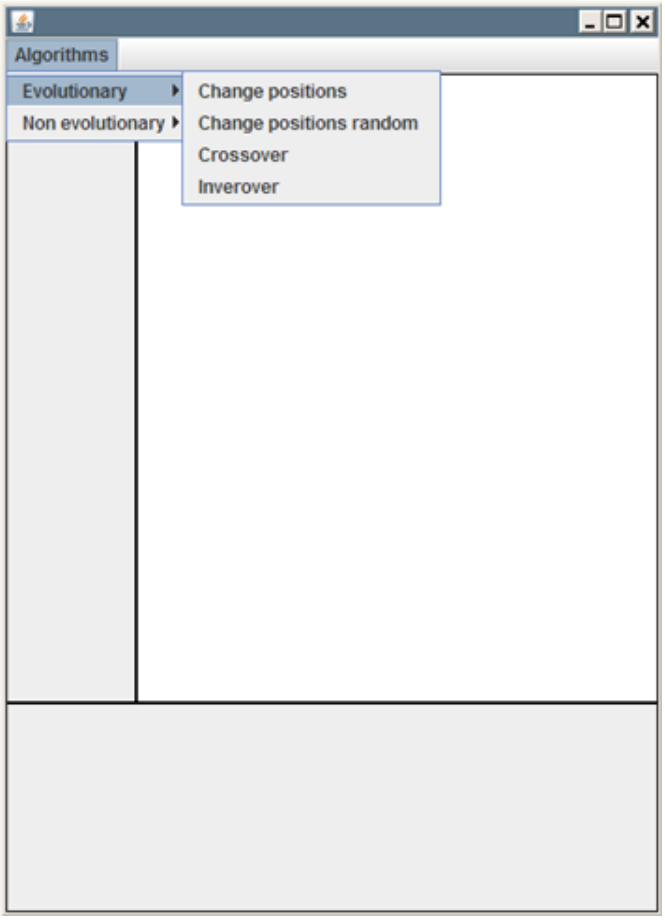
gy=(1 x x 4 x 6 7 x 9)

Ezután egyszerűen bemásoljuk a második szülő városait az üres pozíciókba:

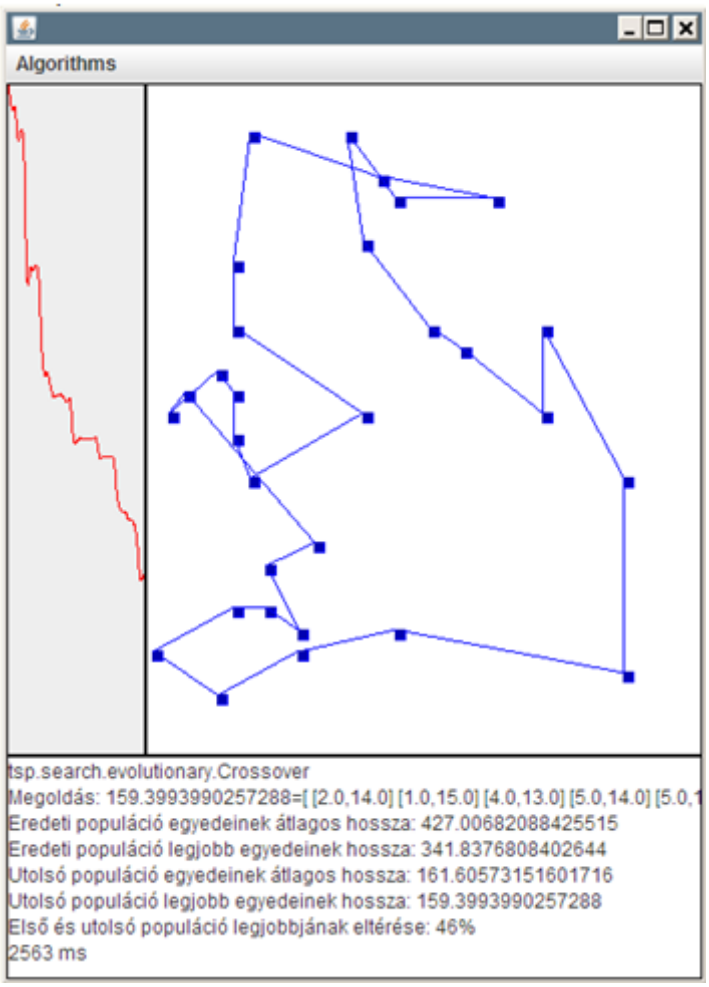
gy=(1 2 8 4 3 6 7 5 9)

6.1.3. A keretrendszer TSP megoldó algoritmusai

Az utazó ügynök problémára és megoldásának szemléltetésére a programom grafikus felülettel is rendelkezik. Nem része a keretrendszernek, csak egy bővítmény. A grafikus megjelenítésért elsősorban az `utils.MainFrame` osztály felel, mely az ablakokat és a főbb komponenseket tartalmazza, mint a menük megjelenítése vagy a gombok. Egy pillanatkép a programról a futásának kezdetekor:



Az ablak közepén a rendszer helyet biztosít az `utils.Drawcities` osztály objektumainak, melyekkel a városokat, valamint az azokat összekötő útvonalat ábrázolja a keretrendszer. Az `utils.GraphFunction` osztály az ablak bal oldalán grafikus kimutatást készít arról, hogy az egyes generációkban a legjobb egyedek fitness értékei hogyan változnak. Ez mindig egy monoton csökkenő függvényhez hasonlít. A keret alsó részén pedig a megoldó algoritmusok futása után különböző információkat jelenít meg, melyekről a 4.1-es fejezetben volt szó. Egy példa a később tárgyalt `Changepositions` osztály megoldására:



Az **Algorithms** menüben kétfajta algoritmuscsoportból válogathatunk, az evolúciós- és a nem evolúciós algoritmusokból. A `tsp.search.evolutionary` csomagban található evolúciós algoritmusok a következőképpen működnek:

- **Changepositions** algoritmus: egy véletlenszerűen megalkotott körútból kiindulva úgy próbálja meg lecsökkenteni annak hosszát, hogy a populáció

minden tagjának véletlenszerűen kiválasztja két városát, majd felcseréli azoknak a körútban elfoglalt pozícióit.

- **Changepositionsrandom** algoritmus: az előző algoritmustól annyiban különbözik, hogy véletlen sokszor választ ki két várost a pozíciójuk felcserélésére, ez egy kicsit távolabb viszi egymástól a szülőt és a gyermeket a megoldástérben.
- **Crossover** algoritmus: kiválaszt két szülőt, mindkettőt kettévágja ugyanazon a helyen, majd az első szülő első részútját a második szülő második részútjával illeszti össze. Az első szülő második részével és a második szülő első részével is ugyanígy cselekszik. A körútban így keletkezhetnek duplikált városok, azokat az algoritmus kitörli és helyettesíti őket olyan városokkal, amelyek kimaradtak a körútból.
- **Inverover** algoritmus: a populáció minden elemére a következőt csinálja: kiválaszt véletlenszerűen egy várost a körútból, majd egy bizonyos valószínűséggel vagy a körút utáni részből választ mellé várost, vagy egy másik véletlenszerűen kiválasztott populációbeli elemben megnézi, hogy melyik város az, amelyik az előzőleg kiválasztott város után következik a túrában. Az eredeti túrában felcseréli a két város közötti sorrendet. A pszeudo-kódja a következő [8]:

1. **for** minden $S_i \in P$-re **do**
2. $S'=S_i$
3. válasszunk ki véletlenszerűen egy *c*-vel jelölt várost *S*'-ből
4. **repeat**
5. **if** *rand*<*p* **then**
6. válasszunk ki egy c' várost S' hátra maradt részéből
7. **else** egy véletlenszerűen kiválasztott *P*-beli egyedben *c*' jelölje a *c* után következő várost
8. fordítsuk meg a sorrendet *c* és *c*' között
9. **if** *eval(S')*<*eval*(S_i) **then**
10. $S_i=S'$
11. **end for**

A *p* érték az implementációmban 20%, azaz ennyi a valószínűsége annak, hogy a *c*'-t az *S*' hátra maradt részéből választja ki. Az *eval* függvény már korábban is szerepelt, a körút hosszát adja meg.

A nem evolúciós algoritmusok közül egy algoritmust implementáltam, melynél a kiindulópont két véletlenszerűen kiválasztott városból álló részút, ezt pedig úgy bővíti, hogy minden városról eldönti, melyik kettő közé tudná beszúrni a legkisebb költség mellett. Ezt a `tsp.search.noevolutionary.MinimalInjection` osztály valósítja meg.

6.1.4. Mérési eredmények

Egy 100 városból álló tesztesetet készítettem az algoritmusok hatékonyságának vizsgálatára, mindegyik algoritmus tízszer oldotta meg ugyanazt a problémát:

Changepositions	
1.	3206
2.	3045
3.	3151
4.	3244
5.	3171
6.	3246
7.	3184
8.	3216
9.	3127
10.	3154
Átlag:	3174

Changepositionsrandom	
1.	4043
2.	4278
3.	4181
4.	4223
5.	4203
6.	4215
7.	4144
8.	4207
9.	4207
10.	4199
Átlag:	4189

Crossover	
1.	2986
2.	3080
3.	2960
4.	3229
5.	3115
6.	2930
7.	3040
8.	2786
9.	2910
10.	2964
Átlag:	3033

Inverover	
1.	3070
2.	2989
3.	3133
4.	3086
5.	3118
6.	3039
7.	2969
8.	3054
9.	3155
10.	2917
Átlag:	3053

MinimalInjection	
1.	858
2.	858
3.	858
4.	858
5.	858
6.	858
7.	858
8.	858
9.	858
10.	858
Átlag:	858

A `MinimalInjection` algoritmus megoldása mind a tíz esetben ugyanaz, hiszen kevés véletlenszerűséget tartalmazó algoritmus, csak a kezdeti két elem kiválasztásában lehetnek eltérések.

A négy evolúciós algoritmus 100 generációval és 50-es populáció mérettel futott le. Végeztem egy másik tesztet 1000 generációval és 100-as populáció mérettel, a futási idő a tízszeresére nőtt, de az eltérés szembetűnő, a legtöbb algoritmusnál majd 70%-os javulást értem el. Az átlaghosszak végeredményben a következőképpen alakultak:

Algoritmus	Átlag hossz
Crossover	1208
Inverover	1288
Changepositions	1699
Changepositionsrandom	2939

Az implementációim alapján az átlaghosszak szerint a következő rangsort állítottam fel:

Helyezés	Algoritmus	Átlag hossz
1.	Lin-Karnighan*	761
2.	MinimalInjection	858
3.	Greedy*	901
4.	Boruvka*	930
5.	Nearest Neighbour*	958
6.	Crossover	1208
7.	Inverover	1288
8.	Changepositions	1699
9.	Changepositionsrandom	2939

A *-gal jelzett algoritmusok eredményeit a Concorde [9] nevű programmal állítottam elő. A probléma optimális megoldása 761, melyet a Lin-Karnighan algoritmus meg is talált. Az általam implementált evolúciós algoritmusok a táblázat utolsó soraiba szorultak ki. A méréseim tehát arra engednek következtetni, hogy speciálisan, az adott problémára megalkotott algoritmusok jóval hatékonyabbak az általános jellegű megoldáskereső módszereknél, így az evolúciós algoritmusoknál is. A generációk és populációk számának változtatásával egymástól nagyon eltérő megoldásokat kaphatunk. Az evolúciós algoritmusok lelke mégis a keresztezési eljárásokban van, ha azokat jól kidolgozzuk és hatékonyan implementáljuk, akkor kevesebb generációval is jó megoldásokat állíthatunk elő.

Összefoglalás

A diplomamunkámban megpróbáltam átfogó képet adni a mesterséges intelligencia kereső algoritmusairól. A számítási intelligencia területéről a neurális hálókat és a fuzzy rendszereket csak átfogóképpen mutattam be, a fő témám az evolúciós algoritmusok voltak. Definiáltam az evolúciós alapfogalmakat, valamint megkíséreltem összegyűjteni azokat a tulajdonságokat, amelyek hatékonyabbá teszi a hagyományos problémamegoldó algoritmusoktól. Az utazó ügynök problémán keresztül részletesen bemutattam, hogyan lehet olyan algoritmusokkal dolgozni, melyek az evolúció folyamatát követik, de erre a problémára hagyományos, heurisztikus eljárásokat is leírtam. A bonyolultabb eljárásoknál pszeudókód segítségével írtam le az algoritmusok működését, némi magyarázattal is.

Olyan keretrendszert fejlesztettem ki, mely az evolúciós algoritmus segítségével bármely problémára megoldást nyújthat, ha azt megfelelően implementáljuk. Az utazó ügynök problémára grafikus megjelenítést is írtam, mely magát a problémát és arra valamely algoritmus által adott megoldást szemlélteti. Méréseket végeztem annak megállapítására, hogy az implementált algoritmusaim közül melyek a leghatékonyabbak. A Lin-Kernighan algoritmust nem sikerült teljes mértékben megírnom, hiszen egy nagyon összetett és nehezen implementálható eljárásról van szó, csak erre egy külön diplomamunkát lehetne szentelni.

Mindvégig az egyszerűségre és átláthatóságra törekedtem a kódolás terén, megpróbáltam alkalmazni azokat a programozási technikákat és technológiákat, melyeket az egyetemi éveim során nyújtott nekem az egyetem.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani konzulensemnek, Jeszenszky Péter adjunktusnak a dolgozat megírásához nyújtott hasznos tanácsaiért, támogatásáért.

Irodalomjegyzék

[1] David L. Applegate, Robert E. Bixby, Vašek Chvátal, William J. Cook, (2006). The Traveling Salesman Problem.

[2] Király Tamás, Szegő László, Kiegészítés az Egészértékű Programozás I. és II. tárgyhoz. http://www.cs.elte.hu/~tkiraly/egeszerteku_jegyzet.pdf

[3] Helsgaun, Keld. (2000). An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. Department of Computer Science Roskilde University DK-4000 Roskilde, Denmark. http://www.akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf

[4] Prof. Dr. Kathrin Klamroth, Branch and Bound Algorithm to solve the TSP. http://www.am.uni-erlangen.de/~klamroth/lectures/networkoptim06/handout2.pdf

[5] Sachin Jayaswal, Department of Management Sciences, A Comparative Study of Tabu Search and Simulated Annealing for Traveling Salesman Problem. www.eng.uwaterloo.ca/~sjayaswa/projects/MSCI703_project.pdf

[6] Zbigniew Michalewicz, David B. Fogel, (2000). How to solve it: Modern Heuristics

[7] Seniw, D. (1991). A Genetic Algorithm for the Traveling Salesman Problem. MSc thesis, University of North Carolina at Charlotte, NC

[8] Guo Tao, Zbigniew Michalewicz, Inver-over Operator for the TSP. http://www.cs.adelaide.edu.au/~zbyszek/Papers/p44.pdf

[9] Concorde TSP Solver. http://www.tsp.gatech.edu/concorde.html

[10] Beardwood, J., J. J. Goode. 1977. The traveling salesman problem: A duality approach. Mathematical Programming 13, 221-237

[11] World traveling salesman problem. *www.**tsp**.gatech.edu/**world***