Téma Beszámoló

Útvonal és erőforrás optimalizáló rendszer készítése

Tagok:
Holló-Szabó Ákos
Koppány Bence
Manninger Miklós
Réti Marcell

A Téma leírása:

Célunk egy térképen, vagy ehhez megfelelő gráf struktúrában (ahol a csúcsok a városok, és az élek a városok között futó utak) való összes város, csúcs meglátogatása minél optimálisabb idő alatt. A matematikában, ezt utazó ügynök problémának nevezik.

Az utazó ügynök problémában a bemenetünk egy teljes gráf és a várt eredmény pedig egy lehető legminimálisabb összsúlyú Hamilton körút, azt szimulálva, hogy az ügynökünk minden csúcsot végigjárt egyszer, és ezt megpróbálta a leggyorsabban megtenni. A probléma NP-teljes számítási nehézségű, ami annyit tesz, hogy egyelőre nem találtak rá polinom időben lefutó algoritmust, nem determinisztikusan polinom időben megoldható. Ha egy NP – teljes problémára, (amely minden NP-beli problémánál nehezebb) egy polinom idejű optimális algoritmust találna valaki, az megoltaná a P=NP? híres matematikai kérdést és teljesen megváltoztatná a matematikai hozzáállást jó néhány témakörből.

A mi feladatunk felkutatni és leimplementálni a legjobb approximációs módszereket, amik természetesen nem az optimális megoldást adják, csak egyre jobb lefutási időt vagy egyre jobb becsléseket, közelítést adnak.

Készítenünk kellett egy vizualizációs keretrendszert, ahol a leimplementált algoritmusokat vizsgálhatjuk meg, akár futás közben, illetve egy hozzá tartozó teszt keretrendszert, ahol a futási eredményeket tudjuk kiértékelni.

A probléma bonyolultságát redukálandó, teljes gráfokat használtunk, ahol minden csúcs mindegyik másikkal egyszeresen össze van kötve, illetve betartottuk, hogy a gráf bármely három pontjára igaz a háromszög-egyenlőtlenség tétele. Ezzel a kikötéssel redukáltuk a problémát (ezt euklideszi utazó ügynök problémának nevezik), bár ezzel a bonyolultsága nem csökkent, hiszen már a Hamilton kör keresése is NP – teljes probléma. Az euklideszi tér jellemzőit viszont kihasználhatjuk olyan módon, hogy a gráftérbeli feladat bármikor átültethető egy térképen értelmezett valós problémába.

A feladat komplexitását növeljük azzal, hogy nem egy ügynököt, hanem tetszőleges számú ügynököt indítunk a gráf csúcsainak legkevesebb idő alatti bejárása érdekében, ezzel még jobban megközelíthetünk egy való életbeli problémát. Természetesen az ügynökök és a gráf paramétereit mi generáljuk a rendszerbe.

A vizualizációs keretrendszer:

A rendszer alapvető elgondolása az volt, hogy átfogó keretet adjon az implementálandó algoritmusoknak. Rendelkezzen egy felülettel, ahol a gráf és ágensek információit lehet bevinni a rendszerbe, illetve ezeket el is lehessen menteni. Továbbá szükséges volt egy grafikus rendszer kialakítása, amely az adott algoritmus szerint jeleníti meg a folyamat lépéseit.

Felhasználás:

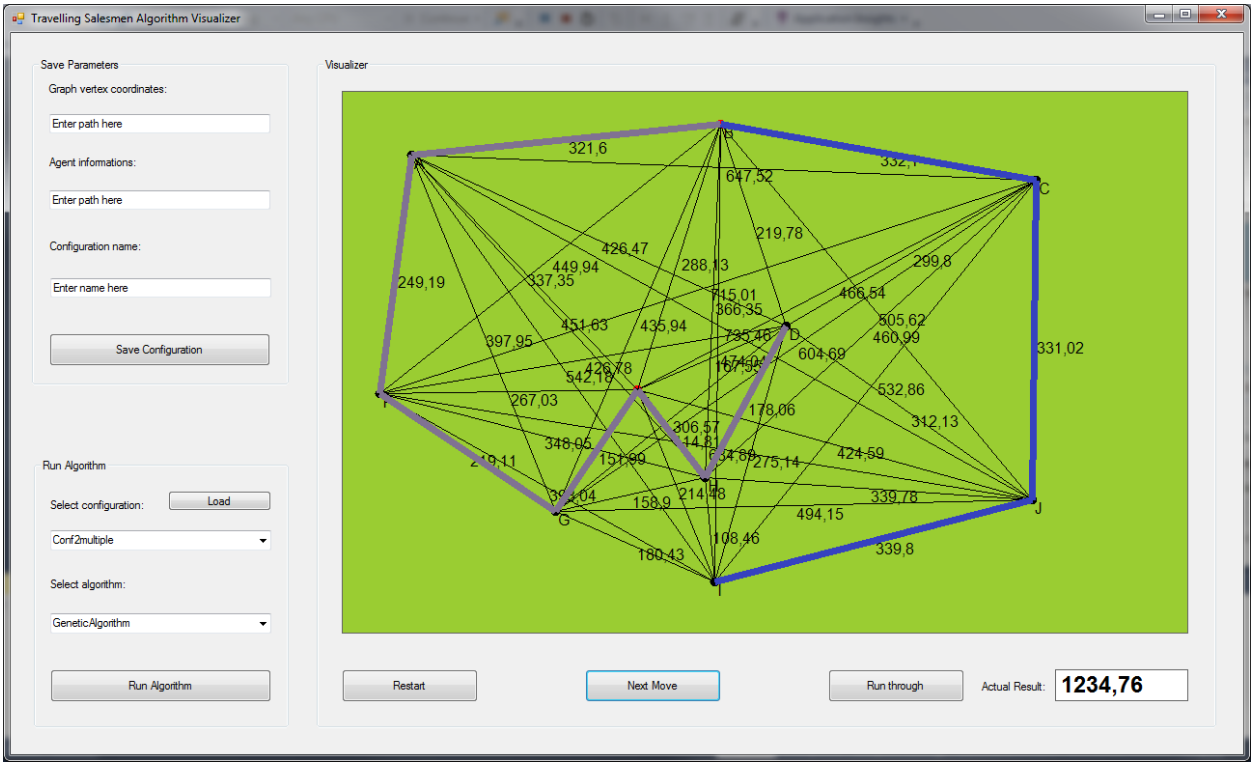
Az adatok beolvasása szöveges állományból történik. A gráf koordinátáit adhatjuk meg az alkalmazás megjelenítő egységének két dimenziós koordinátarendszerében megjelenítve, illetve az ágensek kezdőpozícióját, hogy melyik ügynök mely indexű csúcsból indul. A legtöbb algoritmus egyelőre úgy lett megírva, hogy az első ágens kezdőpozíciójától indul a bejárás minden ágensnek, így a többi kezdő index elhanyagolható.

Az adatok beolvasása után lehetőségünk van elmenteni azokat egy sorosítható összefoglaló konfigurációba, ami tartalmazza a gráf és ágensinformációkat egyaránt. Ez az objektum később vissza is kérhető, újra betölthető.

A konfiguráció kiválasztása után ki kell választani a futtatandó algoritmust, majd elindítani azt. Az algoritmusok lefutási, számítási ideje nagyban függ a gráf pontok és ügynökök számától, így elképzelhető, hogy még a kezdeti inicializációs lépés feldolgozása is hosszadalmas lehet.

Ha az algoritmus elindult, akkor lehetőségünk van lépésenként futtatni és kiértékelni a megoldást, vagy végigfuttatni az algoritmust az algoritmus kilépési feltételéig. A végigfuttatjuk a programot az algoritmuson, akkor lehetséges, hogy az egyes lépések vizualizációját nem látjuk rendesen, ha gyorsan vált a rendszer a lépések között. Az algoritmus tetszés szerint újraindítható, illetve bármikor válthatunk az algoritmusok között.

Az 1. ábrán látható a felhasználói felület a fentebb felsorolt funkcionalitásoknak megfelelően. A bal felső csoport felelős a gráf információk és ágensinformációk fájlból való betöltéséért és konfigurációként való elmentéséért. Az alatta lévő csoport keretein belül választhatjuk ki a kívánt konfigurációt és a rajta futtatni kívánt algoritmust. A *Run Algorithm* gomb segítségével inicializálhatjuk és elindíthatjuk a folyamatot. A felhasználói felületen ekkor megjelenik a gráf és az inicializációs lépés eredménye. Ezek után a *Next Move* gomb segítségével léptethetjük az algoritmust, vagy a *Run through* gombbal végigfuttathatjuk azt. Az aktuális legjobb eredményt az *Actual result* mező értékeként láthatjuk.



1. Ábra: Grafikus keretrendszer felhasználói felülete

Implementáció:

Tervezési szempontból az alkalmazás elkülöníthető rétegeket valósít meg, amelyek fejlesztését külön is lehet végezni. Az implementáció Visual Stúdióban készült a C# nyelv és a WinForms keretrendszer segítségével.

A program tartalmaz, olyan alap osztályokat melyekhez mindegyik réteg hozzáférhet. A *Vertex* és *Edge* osztályok egy gráf csúcsait és éleit reprezentálják, a *Coordinate* osztály a való életbeli koordinátázáshoz szükséges. Ezen alap osztályok felhasználásával készült az *AbstractGraph* és ebből leszármazó egyszerű gráf, *SimpleGraph*, és egyszerű teljes gráf, *CompleteGraph*. A gráf osztályok implementálásakor a gráffal végzett műveletek során (például csúcs hozzáadás, él elvétel stb.), a gráf nem lép ki a saját típusából (például a teljes gráf egy csúcs hozzáadása után is teljes gráf marad). Az *Agent* és *AgentManager* osztály tárolja az ágensinformációkat. A konfigurációt, ami a gráf és ágens információkat tartalmazza, a szerializálható *Configuration* osztály tartalmazza.

A felhasználói felület, az aktuális algoritmus és gráf állapot kirajzolásáért a fő ablak felel. Az ablak meghívható függvényeiben megjelennek az előzőleg említett alap osztályok.

Az Algoritmusok ősosztálya az *Algorithm*, amely egy egységes interfészt biztosít az összes leszármaztatott algoritmus, és az őket meghívó folyamatok számára. Minden algoritmus külön osztályba lett kiszervezve, így, ha új algoritmust szeretnénk felvenni, azt leszármazott osztály szinten kell megkódolni. Az implementált algoritmusok osztályai rendre: *BruteForce*, *Christofides*, *GenetcAlgorithm*, *GreedySearch*.

A felhasználó felület és az algoritmusok közötti kapcsolatot a *Coordinator* osztály teremti meg, amely referenciát tárol a megnyitott ablak osztályáról és az aktuálisan futtatni kívánt algoritmusról. A felhasználó általi hívások az ablak interakciójából a *Coordinator* osztályba futnak be, amely meghívja az adott algoritmus megfelelő függvényeit. Miután az algoritmus szakasz lefutott, a Coordinator kinyeri a változásokat és eredményeket, majd meghívja az ablakot frissítő metódusokat.

A fájl kezeléssel kapcsolatos műveleteket a *FileManager* osztály végzi. Feladata a gráf és ágens adatok kinyerése szöveges erőforrásból, illetve a konfigurációk szerializálása és deszerializálása.

Algoritmusok:

Brute Force

A Brute Force (nyers erő) algoritmus a legegyszerűbb algoritmusok közé tartozik, hiszen a legtriviálisabban oldja meg az ügynök problémát, amit azt jelenti, hogy az összes lehetséges útvonalat kiszámítja és közülük a legrövidebbet visszaadja egy adott gráfban, azaz számunkra az optimális útvonalat.

Esetünkben teljes gráfokat vizsgálunk, ami azt biztosítja, hogy bármely két tetszőleges csúcs között biztosan fut egy él. Ezt kihasználva, ha a gráf csúcsait egymás után rakjuk valamilyen sorrendben, akkor azon végig menve egy Hamilton utat/kört kapunk, attól függően, hogy vissza szeretnénk-e térni az indulási pozícióba. Ezt a gondolatmenetet folytatva, ha egy gráf csúcsait permutáljuk, akkor az összes lehetséges bejárási sorrendet megkapjuk, amelyekre igaz, hogy minden csúcsot pontosan egyszer érintettünk. Egy n csúcsú gráf esetén az n! darab permutációt jelent. A brute force algoritmus egyik nagy problémája, hogy mivel az összes lehetséges esetet megvizsgálja, így nagyobb gráfok esetén ez a megoldás szinte kivitelezhetetlen a hosszú futási idő miatt. Azt is tudjuk, hogy egy teljes gráfban (n-1)!/2 darab különböző Hamilton-kör van. Ebből jól látszik, hogy az algoritmus ugyan azt a kört többször is megtalálja és kiszámolja rá az út értékét, tehát nem csak lassú, de rengeteg ismétlést végez feleslegesen.

Ezek után jogosan merül fel a kérdés: Miért implementáltuk a brute fore algoritmust? Annak érdekében, hogy a többi implementált algoritmust (lásd lentebb) által adott eredmények jóságát tudjuk mihez mérni, szükségünk van az optimális megoldásra is. Ez a későbbiekben jó viszonyítási alapot, támpontot ad ahhoz, hogy össze tudjuk hasonlítani a többi algoritmus által adott eredményekkel (pl futási idő, megoldás pontossága).

Az eddigi leírt megoldás egyelőre csak egy ágens létével foglalkozik. Annak érdekében, hogy egyszerre több ágenst is be tudjunk vetni, a fenti megvalósítás némi módosítást igényel. Ebben az esetben meg kell határozni, hogy melyik ágens melyik csúcsokat fogja bejárni és milyen sorrendben. Ennek szemléltetését az alábbi példa mutatja.

Csúcsok száma: 8.

Ágensek száma: 2.

A csúcsokat számokkal különböztetjük meg: 1 2 3 4 5 6 7 8

Az ágenseket az egyszerűség kedvévért betűvel különböztetjük meg: a és b.

Egy lehetséges csúcs sorrend bejárás: 1 3 4 2 6 5 8 7.

Az ágensek egy lehetséges hozzárendelése: a a b b a a b a.

Összegezve:

a ágens által bejárt csúcsok: 1 3 6 5 7.

b ágens által bejárt csúcsok: 4 2 8.

Az összes ágens-csúcs kisosztást úgy kapjuk meg, ha vesszük az ágensek ismétléses permutációját a gráf csúcsszámának függvényében. Ez x^n darab megoldás, ahol n a csúcsok számát jelöli, x pedig az ágensek számát. A brute force algoritmus több ágens esetén tehát egy permutációból és egy ismétléses permutáció egymásba ágyazásából áll. Fontos még megjegyezni, hogy Hamilton utakat keresünk, vagyis nem kötelező a kezdőpontba való visszatérés, továbbá az ágensek kezdőpozíciója megegyezik, így más-más pontokból indítva őket más-más eredményt fogunk kapni.

Christofides

A Cristofides algoritmus volt az első algoritmus, amellyel foglalkoztunk a projekt folyamán. Ugyan még csak az egy ügynök problémára jelentett megoldást, de mivel az egy ügynök problémának a több ügynök probléma speciális esete, jelentősen elősegítette a probléma feltérképezését, megértését, és a csapat összehangolódását. Fontos szerepet játszott abban is, hogy elkészüljön a keresztrendszer, amiben a további algoritmusokat futtattuk, és teszteltük.

A Christofides ugyan csak egy közelítő módszer, de jól megírva rendkívül gyorsan ad páratlanul jó közelítéseket az egy ügynök problémára. Az alap ötlete az, hogy ha veszünk egy minimális súlyú feszítő fát, és azt a lehető legkisebb súlyú élekkel Euler körré alakítjuk, akkor egy olyan él halmazt kapunk, ami a legnagyobb éleket nem tartalmazza, és amelyet könnyű olyan Hamilton úttá alakítani, melynek élei a legrövidebbek közül valóak. Ez a Hamilton út jelentette a teljes gráfban az egy ügynök probléma megoldását.

Az algoritmus által specifikált lépések összefoglaló jelleggel:

1. Először keressünk egy minimális összsúlyú feszítő fát a térképet reprezentáló teljes gráfban
2. Keressük ki a fa páratlan fokszámú éleit, és készítsünk egy teljes részgráfot belőlük és a köztük futó élekből.
3. A teljes részgráfban keressünk minden csúcsot lefedő, minimális összsúlyú független él halmazt.
4. A független él halmaz csúcsait a fában a megfelelő csúcsokkal megfeleltetve fésüljük össze a két gráfot. (A mindkettőben szereplő éleket itt duplikálni kell)
5. Ekkor egy olyan gráfot kaptunk, aminek van Euler köre, mivel minden csúcsának páros a fokszáma. Ennek az az oka, hogy a független él halmaz élei a fa páratlan fokszámú csúcsainak fokszámát eggyel növelték, a párosoknak pedig egyetlen nem üres részhalmazát se fedik. Keressük meg ezt az Euler kört!
6. Az Euler körből hagyjuk el az ismétlődő csúcsokat úgy, hogy minden csúcs pontosan egyszer szerepeljen végül. Ekkor egy Hamilton kört kapunk, amiből ha elhagyjuk az egyik az ügynök központra illeszkedő élet, akkor meg is kapjuk a keresett Hamilton utat.

A feladat megoldásának pontossága leginkább ebben az utolsó lépésben dől el, mivel a többi lépésben többnyire jól ismert algoritmusokat kellett alkalmazni, melyeknek közel egyértelmű az eredménye. Ennek a lépésnek viszont számos megoldása van. Érdekességképp azt is megemlítem, hogy bizonyítható, hogy a helyes megoldás is kihozható még ekkor; a probléma csak az, hogy exponenciális futási idővel. Mi ezt a lépést nem optimalizáltuk le teljesítményre, hogy időt nyerjünk a valódi feladatunk megvalósításához.

A lépések és az azokra alkalmazott algoritmusok részletes kifejtése:

1. lépés:
   A minimális súlyú feszítőfa keresésére számos megoldás van, mi egy mohó algoritmust alkalmaztuk, mely bizonyítottan a legkisebb súlyú feszítőfát adja eredményül. Ha több ekkora súlyú feszítő fája van, a gráfnak az nem okoz hibát az algoritmus futásában, egyetlen ilyen fát talál meg. A mi szempontunkból irreleváns az, hogy melyik fát kapjuk eredményül, a súlya számít nekünk. Az algoritmus lényege, hogy számon tartjuk azokat az éleket, melyek egyik végpontja fabéli a másik viszont nem fabéli, az ilyen éleket egy „köztes" nevű halmazba soroljuk. A kiinduláshoz egyetlen tetszőlegesen választott csúcs kell. Ez kivesszük a nem fabéli csúcsok halmazába, és betesszük a fabéli halmazba. Egy ilyen művelet elvégzése után az éleket újra megvizsgáljuk, és frissítjük a „köztes" élhalmazt, mert ha változott a csúcshalmaz, akkor a „köztes" élek is változni fognak. Az élhalmaz újraszámítása után megkeressük a legkisebb súlyú „köztes" élet, és ez lesz a feszítőfánk következő éle. Egy ilyen élnek van egy fabéli és egy nem fabéli csúcsa. Az utóbbit kivéve a nem fabéliek közül, megkapjuk a következő csúcsot. Ezt betesszük a fabéli csúcshalmazba és újra számoljuk a „köztes" élhalmazt. Addig ismételjük mindezt, amíg el nem fogynak a nem fabéli csúcsok. Ekkor egy csúcshalmaz és egy élhalmaz eredményeképpen megkaptuk a minimális súlyú feszítő fát.
2. lépés
   A megkapott feszítő fa páratlan fokszámú csúcsait kell megkeresnünk. Ez egyszerű feladat volt, hiszen csak megszámoltuk minden csúcsba a befutó éleket, és kiválogattuk a páratlan fokszámúakat. Ezután egy új teljes gráfot hoztunk létre a kapott csúcsokból.
3. lépés:
   Először keressük ki azokat az éleket a vizsgált teljes gráfból, melyeknek függetlenek minden náluk kisebb súlyú éltől. Minden ilyen élhez egy kiválasztási árat rendelünk, amit a következőképp számolunk ki: Vesszük az összes csúcsot az él két végpontja kivételével, és vesszük a még használható élek közül az arra illeszkedő leg kisebb súlyút, majd vesszük azok közül is a legkisebb súlyút, amelyek nem szomszédosak azzal az éllel, aminek az árát számítjuk. Vegyük minden csúcsban az először és a másodszor kiválasztott élek súlykülönbségének abszolút értékét. Ezek összege lesz az adott él ára. Vegyük azt az élet, aminek a legkisebb az ára és keressük meg azt a két lehető legkisebb súlyösszegű élet, ami szomszédos vele, de egymással nem, illetve vegyük az adott él és a tőle független élek közül a legkisebb súlyú súlyösszegét. Ha az előbbi összeg kisebb mint az utóbbi, akkor a két vele szomszédos élet választjuk ki, ha nem, akkor meg az adott élet. A kiválasztott élet bele helyezzük az eredmény élhalmazba, és szomszédjaival és végpontjaival együtt eltávolítjuk a vizsgált teljesgráfból, ami ekkor is teljes maradt. Ezeket a műveleteket

iteráltastjuk, amíg végül hat vagy egy él marad. Ha egy maradt, akkor azt kiválasztjuk, ha hat, akkor pedig azt a kettőt, amelyeknek kisebb a súlyösszege. Az eredmény halmazba helyezve az utolsó kiválasztottakat, az elkészült.

4. lépés:
   Az első lépés eredménye egy élhalmaz és egy csúcshalmaz volt, a harmadik lépés eredménye pedig egy élhalmaz. Ezt a két élhalmazt összegezve kapunk egy gráfot, mely az eredetinek egy része. Az „összeolvasztás” során dupla élek is keletkezhetnek, de ez nem jelent problémát, hiszen a következő lépésben egy Euler kört keresünk.
5. lépés:
   A kapott gráfban, fent említett tények miatt, biztosan létezik Euler kör. Az Euler kör keresésére a Hierholzer algoritmust használtuk. Ez az algoritmus két csúcslistával és két éllistával dolgozik: Egy futás közbeni, úgynevezett pálya csúcslistával és a végeredmény, az úgynevezett kör listájával, valamint egy használatlan és egy használt éllistával. A pálya és kör listákban fontos a sorrend, mert az reprezentálja a kört, az egymásutániságot. Az algoritmus irányított gráfra működik, de nekünk tökéletesen megfelel. Az általunk használt élek mindkét irányba irányítottak.
   Az algoritmus lépései:
   a. A használatlan élhalmaz az összes élet tartalmazza, a használt éllista üres. Egy él használtságának beállítása alatt azt értjük, hogy beletesszük a megfelelő halmazba. A kör halmaz üres halmaz, a pálya halmaznak adjunk egy tetszőleges csúcsot. Egy csúcsot mindig lemásoljuk, és úgy tesszük bele a kívánt listánkba, tehát mondhatjuk, hogy nem is a csúcsokkal dolgozunk közvetlenül, hanem a másolataikkal.
   b. Válasszuk ki a következő csúcsot. A pálya lista utolsó elemének használatlan élei közül válasszuk egyet. Ezt az élet állítsuk használtra, és a másik végén lévő csúcsot adjuk a pálya lista végére.
   c. Folytassuk a b) lépést addig, amíg a pálya lista utolsó csúcsának van használatlan éle.
   d. Ha a használatlan éllista üres, akkor vegyük ki a pálya lista utolsó elemét és adjuk a kör lista végére, addig, amíg el nem fogy a pálya lista. Ekkor készen vagyunk.
   e. Ha a használatlan éllista nem üres, akkor ez azt jelenti, hogy körbeértünk, de nem használtuk fel minden élet. Ekkor a pálya utolsó csúcsát vegyük ki a pálya listából és adjuk a kör lista végére. Ugorjunk a g) lépéshez.
   f. Vizsgáljuk most meg a pálya utolsó csúcsát. Ha nincsen használatlan éle, akkor menjünk az e) lépéshez, ha van, akkor a b) lépésnél folytassuk.
   g. Vége az algoritmusnak, az Euler kört a kör csúcslista tartalmazza olyan formában, hogy az egymás utáni csúcsok között futnak az élek.
6. lépés:

   Az utolsó lépés a Hamilton kör kialakítása. Az fentebbi összefoglalóban már részleteztük és megmagyaráztunk azt, hogy itt miért használunk egyszerű lépést. Ebben a lépésben annyi a feladatunk, hogy végig megyünk a csúcslista sorozaton, és amelyik csúcs szerepelt már, azt egyszerűen kidobjuk. természetesen ügyelünk arra is, hogy az első csúcs kétszer szerepel, hiszen úgy kapunk az útból kört, ha az eleje és a vége ugyan az. Tehát ezt külön lekezeljük. A csúcs elhagyása nekünk nem okoz semmilyen problémát, hiszen az eredeti gráfunk teljes gráf volt, így az elhagyott előtti és az elhagyott utáni csúcsok között is lesz él

az eredeti gráfban. A kapott csúcslista egymás utáni bejárása (pl a lista harmadik csúcsától a negyedikbe megyünk, onnan az ötödikbe, stb.) a keresett Hamilton kört eredményezi.

Mohó algoritmus

A programozásban megismert mohó algoritmusok ismertetőjele, hogy tulajdonképp gondolkozás nélkül, Trial & Error módon, véletlenszerűen próbálkozva próbálják elérni az optimális, vagy az optimálishoz minél közelebbi megoldást. Ez jelen esetben sincsen másképp.

Az algoritmus bemenetei a következőek:

- A gráf: ezen keressük a lehető legjobb megoldást. Jelen esetben úgy vettük, hogy a megoldás jóságát mérő szám, az az, hogy mely ágens járta be a leghosszabb utat. Mivel egy egység megtétele egyenlő a rá fordított idővel (egy egység megtételéhez szükséges idő konstans egy egység), így az adott megoldás megfogalmazása a következőképpen is történhetne: a megoldás jóságát az összes csúcs bejárási ideje adja, ezt szeretnénk minimalizálni. Mivel az ágensek parallel futnak, így az adott megoldás értéke a leghosszabb ideig futó ágens futási időtartama.
- Az ágensek száma: hány ágens fogja megpróbálni egyidejűleg bejárni a gráfot
- Az ágensek kezdőpontja: Minden ágens egy kezdőpontból indul, és az algoritmus nem várja el a kezdőpontba való visszatérést, tehát diszjunkt Hamilton utakról beszélhetünk, minden ügynök esetében.
- PATIENCE_PARAMETER: Ez a bemeneti paraméter mondja meg, hogy egy adott lokális minimumba való ragadás során hányszor próbálkozzon a rendszer az onnan való kilépésből.
- NUMBER_OF_RUNS: Egyszerű generáció szám, hányszor próbáljon az algoritmus új legjobb megoldást keresni az előző legjobb alapján.
- MAX_ROUTE_LENGTH_PER_AGENT: Egy adott ágens maximálisan befutható csúcsait korlátozza. Ezen paraméter megválasztásakor ügyelni kell, hogy a feladat megoldható maradjon!

Az algoritmus kezdeti lépése, hogy létrehoz, egy teljesen randomizált megoldást. A megoldás struktúrája az ügynökök szerint van felbontva. A megoldás minden ügynökre tárol egy tömböt, amiben a gráf csúcsainak indexei vannak. Az ágens tömbjében lévő csúcsindexek azt jelképezik, hogy az adott ügynök abban a sorrendben bejárja az adott csúcsokat. Így elképzelhető az is, hogy egy ügynök el sem indul.

Az algoritmus mindig számon tartja a globális legjobb megoldást és az adott generáció legjobb lokális megoldását. Miután az inicializálás megtörtént megkezdődik egy újabb generáció legyártása, ahol egy az eddigi legjobbnál jobb eredményt szeretnénk legenerálni.

A lokális legjobb megoldást véletlenszerűen kell legenerálni, majd ennek egy úgynevezett szomszédját kell létrehozni. A szomszéd létrehozásának öt metodikája van, amiből véletlenszerűen kell egyet kiválasztani.

Szomszéd generálási lehetőségek:

- Úton belüli inverzió: Egy véletlenszerű úton egy rész utat invertálunk.
  2 3 4 5 6 -> 2 6 5 4 3
- Úton belüli csere: Egy véletlenszerű úton két rész utat felcserélünk egymással.
  2 3 4 5 6 -> 2 4 5 2 3
- Úton belüli beszúrás: Egy véletlenszerű úton egy rész utat máshova szúrunk be az úton belül. ↘
  2 3 4 5 6 -> 2 4 5 3 6
- Utak közötti csere: Két véletlenszerű út egy-egy szakaszát átcseréljük.
  2 3 4 5 6 -> 2 7 6
  7 -> 3 4 5
- Utak közötti transzfer: Két véletlenszerű utat választunk. Az elsőből egy véletlenszerű szakaszt kivágunk és átmásoljuk a másikba, arra a helyre ahonnan az eredetiből indult.
  2 3 4 5 6 -> 2 6
  7 -> 7 3 4 5

Az adott szomszéd generálása után megnézzük, hogy a kijött megoldás jobb-e mint a lokális legjobb, ha igen akkor a lokális megoldást felül írjuk vele. Ha az eddigi legjobb globális megoldásnál is jobb azzal is ugyanezt tesszük. Ezek után a szomszédgenerálás újra indul.

Ha a szomszédok egy idő után nem generálnak jobb eredményt, mint a lokális legjobb akkor egy lokális minimumba érkeztünk. A PATIENCE_PARAMETER szabja meg, hogy hányszor próbálkozzon kilépni belőle. Ha nem sikerül a generáció legyártása a végéhez ért.

A generációk számát a fent említett NUMBER_OF_RUNS szabályozza.

Az algoritmus tehát a leírtak alapján láthatóan nem gondolkozik, csak véletlenszerű cserékkel, inverziókkal, beszúrásokkal próbál utat módosítani és ezzel jobb megoldásokat találni. Nagy előnye, hogy elég gyorsan lefut. Az algoritmus hatékonysága növelhető lehetne szimulált lehűtés bevezetésével.

Genetikus algoritmus

A genetikus algoritmus nem konkrétan egy feladatra alkotott algoritmus, mint a Christofides, hanem egy általános módszertan problémák közelítéses megoldására. Előszeretettel használják NP-teljes problémák esetén, mivel rendkívül kedvező futási idővel ad nagyon jó közelítést, és nem is tartozik a legbonyolultabbak közé (, még ha jobb teljesítményt hozó variánsairól és más algoritmusokkal alkotott hibridjeiről ez nem is mondható el feltétlen). Nevét azért kapta, mert az evolúció modelljének mintájára alkották meg, felfogható egy fajta nemesítés ként is. Itt a fajunk nem más, mint a konkrét feladat kellő mértékű általánosított (enyhített) megoldáshalmazának részhalmaza. A cél pedig hogy a megoldáshoz legközelebb állók életben maradjanak, és nagyobb valószínűséggel szaporodjanak, mint rosszabbul teljesítő társaik.

A genetikus algoritmusok során be kell vezetni néhány alapfogalmat, amit a további leírásban és az algoritmusban is használni fogunk.

- Kromoszóma: A feladat enyhítésének egy megoldása. Jelen esetünkben ez az enyhített feladat a következő:
    - az ügynökök mindegyike járjon be egy csúcsot legalább
    - minden csúcsot csak egy ügynök érintsen
    - minden csúcsot érintsenek az ügynökök együttvéve.
- Allél: A kromoszómáinkat megkülönböztető elemi attribútumok. Jelen esetben ez az, hogy melyik ügynökök, melyik csúcsokat, milyen sorrendben járják be.
- Populáció: Kromoszómáinknak halmaza. Ők azok, akik a nemesítésben részt vesznek. Létszámukat érdemes konstans értéken tárolni a konstans memóriaigény biztosításának érdekében
- Fitness: Annak mértéke, hogy egy adott kromoszóma milyen közel áll az eredeti probléma megoldásához. Ez alapján válogatjuk őket. Jelen esetben ez a leghosszabb út, amit egy ügynök megtesz, hiszen az MTSP-ben ennek kell a lehető legrövidebbnek lennie, mást nem is általánosítottunk.
- Generáció: A nemesítés több iteráción keresztül tart. Az egy iteráció végére megmaradt populációt szokták az iteráció sorszámával is jellemezni, és rajtuk keresztül a genetikus algoritmus hatékonyságát.

Egy általános genetikus algoritmus három főbb lépésből áll, melyek során generációról generációra lépve újabb populációt állít össze. Az Inicializáció csak az algoritmus előtt fut le. Azt követően a másik három lépést írtam le abban a sorrendben, ahogy az iterációban is követik egymást:

- Inicializáció: Létre kell hoznunk egy kezdeti populációt, ami a nemesítés kezdeti alanyául szolgál. Mivel nem ismerjük a megoldást, ekkor még nem tudjuk milyen allélekből áll, de ahhoz hogy az algoritmus jól működjön, elengedhetetlen, hogy legtöbb allélje kellő mennyiségben jelen legyen a kezdetleges populációban. Ezt csak valószínűségi alapon tudjuk biztosítani azzal, hogy nagyra vesszük a populáció méretét, és biztosítjuk sokszínűségét.
- Szelekció: Minden populációbeli egyednek megvizsgáljuk a fitness értékét és kiválasztjuk a legjobb, legéletrevalóbb kromoszómákat. A kiválasztottakból származtatjuk alapjáraton az új generáció maradékát, de ekkor még fent áll annak az esélye, hogy benne ragadunk egy helyi minimumban. Ez alatt azt kell érteni, hogy ha egyik körben az egyik szükséges, de nem jelentőségteljes allél nem jelenik meg a legjobbaknál, akkor elveszhet, és ugyan nagyon közel kerülünk a célunkhoz, de nem tudjuk elérni. Ennek orvoslására a legjobbak közé még beválogatunk pár rosszabbat is véletlenszerűen, így adva több esélyt a szükséges allélok fent maradására.
- Keresztezés: A kiválasztott elemeket megtartva, és a többit elvetve megkaptuk azt a halmazt, amiből az új kromoszómákat származtatni szeretnénk. Ebben a szakaszban történik meg maga a származtatás. Valamilyen elv szerint összepárosítjuk a kiválasztott kromoszómákat, és páronként egy vagy több új kromoszómát származtatunk belőlük. Az újonnan keletkezett, és a megtartott kromoszómák fogják alkotni az új populációt.
- Mutáció: Hogy növeljük az kromoszómák sokszínűségét, a keresztezéssel keletkezett kromoszómák egy részén még variálunk is. Például egy allélt átírunk, vagy két azonos típusú allélt felcserélünk úgy, hogy még mindig megoldása maradjon az általánosított feladatnak. Ha egy a megoldásban szereplő allél elveszik a szelekció miatt, vagy sose létezett, akkor ez az egyetlen esélyünk arra, hogy újra bekerüljön a populáció bármelyik kromoszómájába.

A konkrét megvalósítás:

Az első, amire ki kell térnünk az a kromoszóma adatszerkezete, mivel ezt is sokféleképpen meg lehet valósítani. Jelen esetben a teljes gráf csúcsait számokkal azonosítottuk, és az alléleket két tömbben tároltuk. Az első tömb a csúcsok egy permutációját tárolta, amiben nem szerepelt a kezdőpont, a második tömb pedig azt, hogy melyik ügynök mennyit jár be belőlük. Például ha a hat csúcsról van szó és a permutációnk a {1,4,2,0,3,5}, és két ügynökünk van, melyekre a tömb az {2,4}, akkor az első ügynök az 1,4 csúcsokat járja be ebben a sorrendben, a másik pedig a 2,0,3,5 csúcsokat. Az allél fittségét is maga a kromoszóma tárolta

A populációt mi random generáltuk, először nagy populációszámmal, így biztosítva, hogy minden szükséges allél elégszer szerepel benne. A kromoszómák fittségét az Inicializálás után számoltuk ki, ezt követően kiszelektáltuk az elemek felét. Ennek gyorsítása érdekében először fittség szerinti növekvő sorrendbe rendeztük a kromoszómákat. A kiválasztottakat teljesen véletlenszerűen szerveztük párba a keresztezéshez. (Itt látszik, hogy mivel a populáció felének elemei párba állíthatóak, a populáció méretének oszthatónak kell lennie néggyel.)

A keresztezés talán az egész algoritmus legbonyolultabb és legfontosabb része. Ugyanis úgy kell új elemet generálnunk kettő másikból, hogy az minél nagyobb valószínűséggel járhasson javítással, és ne veszítsék el az elemek pozitív tulajdonságaikat. A két keresztezendő elemet

nevezzük apának és anyának. A módszerünk az, hogy megtartjuk az anya csúcskiosztását az ügynökökre, és bizonyos elemeket rögzítünk a permutációjában, majd az instabil elemeket olyan sorrendbe rendezzük, ahogy az apában is vannak. Ekkor, ha az anya elég jó megoldás, akkor kedvező megtartani az ügynökök csúcs kiosztását. Ahhoz, hogy a permutáció pozitív tulajdonságaiból is megtartsunk, érdemes biztosítani, hogy szomszédos csúcsokat is rögzítsünk, így esélyt adva olyan élek megtartására, ami a cél megoldásban is szerepel vagy köze van hozzá. Ez után származtassunk a párosított két elemből egy kromoszómát fordított szerepkiosztással is.

Jól látható ebből, hogy a keresztezés nem variálja az ügynökök csúcs kiosztását. Nagyon fontos, hogy ez nem is lenne célszerű. Ugyanis keresztezésnél egy olyan kromoszómát akarunk kapni, ami a szüleitől nem esik túl messze, hogy egy hangolás lehessen rájuk nézve. Azért jó ez nekünk, mert ha a szülők közelebb állnak a megoldáshoz, mint az előző generáció átlaga, akkor nagyobb valószínűséggel találunk javításra a közelükben. A csúcs kiosztást viszont, ha variáljuk, akkor a szülőktől nagyon messze eső, tőlük teljesen idegen megoldást kapunk, ami nagyon kis valószínűséggel lesz náluk jobb.

A mutáció során két csúcsot cserélünk fel a permutációban, de itt sem célszerű a csúcsok kiosztásához nyúlni a feljebb említett indokkal.

Genetikus algoritmus paraméterei, optimalizációja:

A genetikus algoritmus hatékonysága, avagy sebessége, memóriaigénye, és a megoldás pontossága több paramétertől függ, és ezeket nagyon jól kell hangolni ahhoz, hogy a kívánt működést érjük el. A paraméterek az alábbiak:

- A csúcsok száma: Magától értetődően minél több csúcsunk van, annál több helyet foglal egy kromoszóma permutációja, és persze annál több permutáció lehetséges, így annál nagyobb populáció kell ahhoz, hogy a keresett megoldás alléljai kellő valószínűséggel megjelenjenek és túléljenek.
- Az ügynökök száma: Az is egyértelmű, hogy miért jár több memóriaigénnyel, és számítással, de talán még fontosabb, hogy a csúcsokat akkor lehet a legtöbbféleképpen kiosztani az ügynökök között, amikor azok száma legközelebb áll a központon kívüli csúcsok számának gyökéhez. Ennek egy elég hosszas számolás a levezetése, de a lényeg az, hogy ismétléses kombináció darab kiosztás létezik, de két megoldás ekvivalens, ha csak a kiosztás sorrendjében különbözik, és a kiosztottaknak megfelelően keverednek a csúcsok is a csúcs permutációban, avagy az ügynökök által bejárt utak páronként megfeleltethetőek. A sima kombináció akkor maximális, ha k=n/2, de ez már az ismétlésesről nem mondható el. Így hát akkor jártunk jobban futási időre és pontosságra, ha nagyon sok vagy nagyon kevés ügynökünk van.
- A populáció mérete: A populáció mérete egy nagyon két oldalú dolog, hiszen minél nagyobb a populáció, annál kevesebb generáció alatt, és annál pontosabb eredményt kapunk, de annál több helyet foglal, és jelentősen több időt emészt fel. Elvégre az elemeket mindig először rendezni kell fittség szerint, ami O(n*log(n)) művelet, és utána a kiszelektálás O(n), a keresztezés is O(n) az ügynökök számára, a mutáció is várhatóan O(n). Így hát végső soron n*log(n) arányos vele a számítási idő egy iterációra.

- A mutáció valószínűsége: Minél nagyobb, annál többször kell elvégezni a mutációt, de végső soron elveszik konstans szorzóként. A memória igénye sem túl kiemelkedő, mivel az általa módosított kromoszómát írja csak át. Sokkal nagyobb hatással van viszont a pontosságra, mivel ha túl magas, akkor túl nagyra növeli a szülők és a gyerekek közötti különbséget, ami -mint korábban is említettem- nem túl jó dolog. Ha viszont nagyon alacsony, akkor nagyon magasra ugrik a fontos allélok kihalási esélye.
- Rosszabbul teljesítő kiválasztottak aránya: Ez a paraméter se a memória igényt, se az egy iteráció számítási idejét nem növeli, de ha túl magas, akkor a pontosság nagyon leromlik. Általában nagyon alacsony kell, hogy legyen, de elengedhetetlen paraméter.
- Megengedett generációk száma: Ezzel egyenesen arányosan nő a számítási igény.

A genetikus algoritmusnál először is nagyon fontos megjegyezni, hogy a populáció mérete, a csúcsok és az ügynökök száma nem változik, így a legstatikusabb tárolók is tökéletesen megfelelnek. A nem kiválasztott elemeket pedig nem szabad törölni, mivel ugyan annyi új elem fog keletkezni. Jobban járunk, ha beléjük mentjük el sorra az új elemeket. Az efféle dolgokkal ugyan egy konstans szorzó erejéig, de rengeteg számítási időt lehet spórolni.

Mik a bevett paraméterezések?

Először is a genetikus algoritmusnak ahhoz, hogy egyáltalán működjön, már szükséges a jó beállítás. Éppen ezért vannak nagyon standard beállítások, amikre mindenképp működni fog az algoritmus:

DeJong Beállítások (From [DeJong and Spears, 1990]):

DeJong beállításai a fő standard, ha genetikus algoritmusról van szó. DeJong bebizonyította, hogy jól működnek akármely problémára, amely Genetikus algoritmus kompatibilis.

- Populáció mérete: 50
- Generációk száma: 1,000
- Mutáció típusa: bit negálás
- Mutáció esélye: 0.1% minden kromoszóma minden bitjére a kromoszómát leíró adatnak
- A rosszabbul teljesítők arányára nem tér ki

Grefenstette Beállítások (From [Grefenstette, 1986]):

Tipikusan akkor használják, amikor a számítógép memóriakorlátozása és korlátozott processzálási sebessége nem enged meg túl nagy populációt.

- Populáció mérete 30
- Generációk száma: nem tér ki rá
- Mutáció típusa: bit negálás
- Mutáció esélye: 1% minden kromoszóma minden bitjére a kromoszómát leíró adatnak
- A rosszabbul teljesítők arányára nem tér ki

MicroGA Beállítások From David L. Carroll

Nem széles körben elterjedt, de használói négyszer kevesebb evolúció igényről számolnak be.

- Populáció mérete: 5
- Generációk száma: 100
- Keresztezés típusa: „uniform”
- Mutáció típusa: „ugrás és csúszás”
- Mutáció valószínűsége: 2% és 4% között

Paraméterválasztás:

A már a feljebb említett algoritmusoknál is megjelent,hogy a kromoszómák által tárolt adat bitértéke szerepet játszik a paraméterek kiszámításában.

Hány bitben fejezhető ki egy kromoszómánk információtartalma?

$csúcsszám!$ darab permutáció lehetséges és $\binom{\text{csúcsszám+ügynökszám}-1}{\text{ügynökszám}}$-nál kevesebb súly kiosztás. Így elég jó felső becslés az $1 = \lceil \log(n!) \rceil + \lceil \log\binom{n-m+1}{m} \rceil$, ahol n a csúcsok száma és m az ügynökök száma. Azonban az előbbi képlet nem felel meg nekünk túl nagy csúcsokra, mert 100!-t nem tudunk ábrázolni, így kénytelenek leszünk a következő, durvább becsléssel élni: l=log(n)*(n+m).

Mivel a mi állapotterünk túl nagy, nem lesz elég a standard populáció mérete, így használjuk a következő közelítést:

$$p = \frac{\text{length} * 2^c}{c},$$

ahol a *p* a populáció mérete, a *length* egy kromoszóma mérete bitekben, és c pedig félrevezetően egy tulajdonság átlagos mérete: $\lceil \log(n!) \rceil + \left\lceil \frac{\binom{n-m+1}{m}}{n+m} \right\rceil$, vagy durvább becsléssel: log(n).

A mutációról nagyon kevés irodalmat találtunk, de itt is megfogalmazódott egy logika, amit többen is kifejtettek. Nagyobb mutációs esélyre van szükség olyan példa esetén, ahol a fontos allélok könnyen elveszhetnek az iteráció folyamán, és kisebbre ott, ahol nagy valószínűséggel azért túl élnek. Mi esetünkben erre egy jó példa a csúcsok kiosztása az ügynökök között. Mi úgy generáltuk az első populációnál, hogy az első ügynöknek sorsoltunk valamennyit úgy, hogy a többinek is juthasson legalább egy csúcs, majd a másodiknak is, és az azutáninak is ugyanígy tettünk a maradékból. Ennek az a következménye, hogy mivel nem biztosítottunk azonos valószínűséget az egyes megoldás osztályoknak, lesznek köztük valószínűbbek és kevésbé valószínűek. Például hat csúcs és 3 ügynök esetén az {1,1,4}={1,4,1}={4,1,1} valószínűsége 1/4*1/4+1/4*1/4+1/4=0.375..,míg a {2,2,2}-nek 1/4*1/3=0,08333... és az {1,2,3}={1,3,2}={2,1,3}={2,3,1}={3,1,2}={3,2,1}-nek 1/4*1/4+1/4*1/4+1/4*1/3+1/4*1/3+1/4*1/2+1/4*1/2=0.541666 Ez azt jelenti hogy azok a

kiosztások például, ahol az összes ügynökre ugyanannyi csúcs jut, nagyon ritkák lesznek az első populációban. Másik példaképp vegyünk 10 csúcsot négy ügynökre, itt az {1,1,8}={1,8,1}={8,1,1} valószínűsége 1/8*1/8+1/8*1/8+1/8=0.15625, míg a {3,3,4}={3,4,3}={4,3,3}-nek 1/8*1/6+1/8*1/6+1/8=0.166.... Ebből is látható hogy minél nagyobb a kiosztás értékeinek a szórása, annál kisebb lesz a valószínűsége. Ahhoz viszont hogy tudjuk a pontos valószínűséget már ismernünk kéne a megoldást. Közelíteni ugyan még tudnánk, például ha a városok nagy része közel van a központtól, és alig pár darab van távolabb, de azok is más-más irányban, akkor várhatóan nagy lesz a végső megoldásban a csúcsok kiosztásának a szórása, de ez rend kívül bonyolult matematikai feladat. Ezért mi úgy döntöttünk, hogy a mutáció legyen egy általános középérték, ami a legvalószínűbb esetekben működik a legjobban. Ez az érték a 30%/kromoszóma. Mi ezen kívül a standarddal ellentétben allél cserét használunk, mivel ez nagyjából a bit negálás megfelelője, de bit negálással túl könnyen kaphatnánk érvénytelen megoldást. Ezt mi a kereszteződésből születő egyik gyerekre használjuk 60%/kromoszóma eséllyel.

A rosszabbul teljesítőek arányára sem terjed ki az általam talált szakirodalom, mivel nem egy régóta bevett paraméter. Pár kimutatás van rá, de azok is csak statisztikai alapon saccolnak értékeket. Akkor jobb magasabban tartani, ha hemzseg a gráf a keresett megoldás közeli fittségű, de attól nagyon eltérő megoldásoktól. Mivel azonban jelentősen lassítja a helyes megoldáshoz való konvergálást, érdemes nagyon alacsonyan tartani. Ha x%-on van, akkor 50% az esélye annak, hogy (1-x/2)%-nál több kiválasztott kromoszóma lesz a legjobbak közül. Ez azt jelenti, hogy 100% esetén egyenesen végzetes, mert az aktuális legjobb megoldásra 50%, hogy megmarad az adott körben. 50-10% esetén is nagy zavart okoz, mivel a legjobbak alsó fele tartalmazhat olyan allélokat, amik kellenek a végső megoldáshoz, és így csökkenti az esélyt a fennmaradásukra. Így hát érdemes 10% alatt tartani. 10%-ot akkor érdemes használni, ha a keresett megoldás csúcskiosztása nagy szórású, és sok egyformát tartalmazó, és 1%-ot kis szórású és kevés egyformát tartalmazóakra. Ezen belül pontosabb becslés nem mondható.

A generáció határra nagyobb becslés nincs. Statisztikai alapon megmondható, hogy ha jók a paramétereink, és 1000 generáció fölé csúsztunk, akkor legyen szó akár a legszélsőségesebb esetről is, szinte biztos, hogy valamit rosszul paramétereztünk, de nem kicsit. A mi felméréseink alapján az a tapasztalat, hogy 400 generáció alatt mindig megtaláljuk a keresett megoldást.

Tesztkeretrendszer:

Az algoritmusok teszteléséhez szükség volt egy olyan változatra is, amely nem használ grafikus megjelenítést és teszteléseket lehet rajta elvégezni. Ezért elkészítettünk egy konzolos programot is, mely az algoritmusokat a kapott paraméterekkel létrehozza és a végeredményt egy fájlba írja.

Használata:

1. A megfelelő számmal meg kell adni, hogy melyik algoritmust szeretnénk használni
    a. BruteForceSingleAgent – egy ügynökös brute force
    b. BruteForceMultiAgent – több ügynökös brute force
    c. Christofides
    d. Genetic
    e. GreedySearch – mohó algoritmus
2. Ezután adhatjuk meg a gráf fájlját.
3. Majd az ügynökök fájlját.
4. Ezután, ha a mohó vagy a genetikus algoritmust választottuk, akkor annak sajátos paramétereit is megadhatjuk, és megadhatunk ezek közül egyet, mely egy intervallumokon megy végig a futtatott esetek során.
5. Utána a futtatási számot adhatjuk meg, mely azt jelenti, hogy hányszor fut le az algoritmus.
6. Végül a kimeneti fájl nevét adjuk meg.

A keretrendszer a futás alatt kiírja a képernyőre, hogy eddig mennyiszer futott le. Ha lefutott annyiszor, amennyiszer a felhasználó szerette volna, kiírja, hogy végzett és újabb algoritmus futtatását ajánlja fel. A megoldott számításokat egy fájlba menti el, így azokat felhasználva könnyedén tesztelhetjük az algoritmusainkat.

Adatok feldolgozása:

Példák:

Tesztjeinkben megvizsgáltuk, hogy a genetikus algoritmus optimális populáció számmal tényleg optimálisabb megoldást ad-e mint egy másik populáció szám. Az optimális populáció számokat a genetikus algoritmus fejezetben tárgyalt módon számoltuk ki. A nem optimális számok körülbelül az optimális számok duplája.

Példáinkban futtattuk a Genetikus algoritmust optimális populációszámmal és egy nem optimális számmal, valamint a Mohó algoritmust is.

Példa 1:

Cél: Az optimális populációszám vizsgálata, illetve kevés és sok ügynök bejárásának összehasonlítsa.

Gráf:
100 csúcsú teljes gráf

Ügynökök:
1 ügynök és 16 ügynök

Algoritmus:
Genetikus

Optimális populáció szám	Ügynökszám	Eredmény (távolság)	Futási idő (ms)
Igen	1	21359	19305
Nem	1	20913	35471
Igen	16	2754	20249
Nem	16	2747	37425

Jól látható, hogy mindkét esetben, 1 és 16 ügynökre is, a nem optimalizált populációszámmal szinte ugyan azt az eredményt kapjuk, viszont a futási idő majdnem a kétszeresére nőtt.

Az is könnyen leolvasható az adatokból, hogy egy ügynökkel sokkal nagyobb úthosszt kapunk eredményül, de sok ügynökkel gyorsabb a gráf bejárása. Az is látható, hogy a sok ügynök számításához nem is volt szükség sokkal több időre.

Eredmény:

Az optimális számmal szinte ugyanazt az eredményt érhetjük el, mint nem optimális paraméterezéssel, viszont fele annyi időre van szükségünk a számításhoz. Továbbá kevés ügynökkel sokkal hosszabb bejárást kapunk, mintha sok ügynökkel járnánk be a gráfot, és ehhez ugyanannyi számolási idő szükséges.

Példa 2:

Cél: Az optimális populációszám vizsgálata, illetve 50, 100 és 200 csúcsú gráfok bejárásának összehasonlítsa.

Gráf:
50, 100 és 200 csúcsú teljes gráfok

Ügynökök:
4, 5 és 2 ügynök

Algoritmus:

Genetikus és Mohó

Algoritmus	Optimális populáció szám	Csúcsszám	Ügynökszám	Eredmény (távolság)	Futási idő (ms)
Genetikus	Igen	50	4	1319	2813
Genetikus	Nem	50	4	1365	5919
Mohó	---	50	4	2593	598
Genetikus	Igen	100	5	4928	15369
Genetikus	Nem	100	5	4729	31444
Mohó	---	100	5	5002	738
Genetikus	Igen	200	2	25666	90976
Genetikus	Nem	200	2	25334	207167
Mohó	---	200	2	25369	1606

Jól látható, hogy mindhárom esetben a leggyorsabb a mohó algoritmus volt, 100-szor is gyorsabban futott, mint a genetikus futtatások. A leglassabb a nem optimális genetikus volt. A legjobb eredményt többnyire a nem optimális genetikus adta, de az optimális társa szinte semmivel nem maradt el tőle, átlagosan kevesebb, mint 1%-kal rosszabb eredményt adott, míg a mohó algoritmus nagyobb mértékben eltért ettől.

Eredmény:

Itt is kimutatható, hogy az optimális populációszám jobb, mint más populációszám.

Beláthat az is, hogy kevesebb csúcsra rosszabb eredményt ad a mohó algoritmus, de a csúcsszámok növelésével egyre jobban közelíti a genetikus eredményét. A mohó algoritmus talán egyetlen előnye, hogy akár 100-szor is gyorsabb lehet, mint a genetikus.

Példa 3:

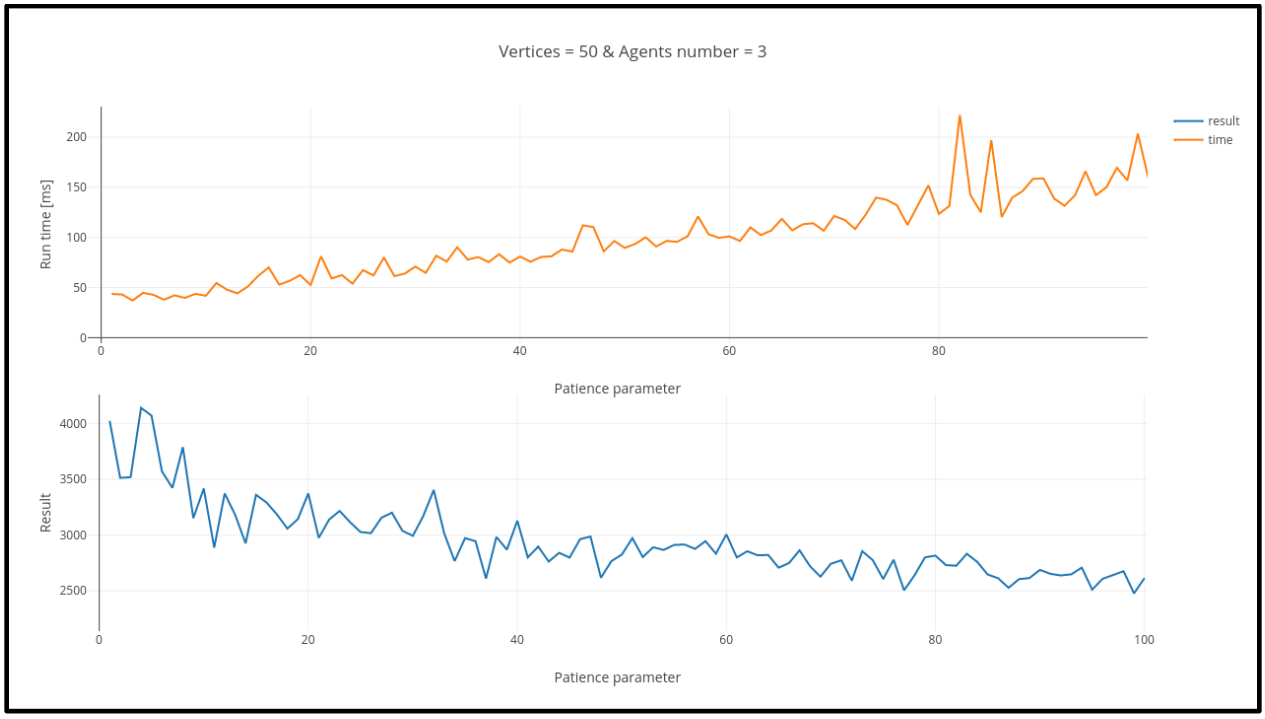
Cél: Annak kimutatása, hogy a mohó algoritmus Patience paraméterének növelésével a futási eredmény javulni, ellenben a futási idő növekedni fog.

Gráf:
50 csúcsú teljes gráf

Ügynökök:
3 ügynök

Algoritmus:

Mohó



Eredmény:

Az adatokat feldolgozva, a diagrammról leolvasható, hogy a Patience paramétert 0-tól 100-ig futtatva, az eredmény tényleg javult (kék vonal), de a számításhoz szükséges idő (sárga vonal) növekedett.

Példa 4:

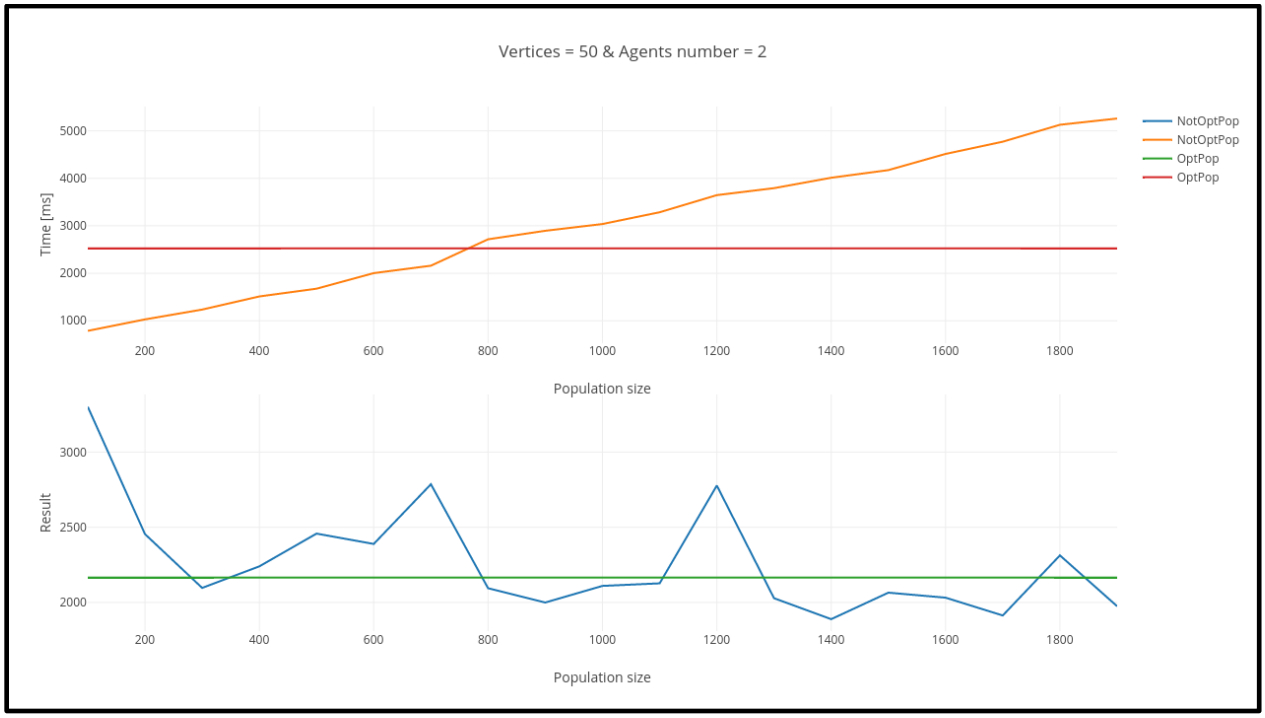
Cél: Annak kimutatása, hogy a genetikus algoritmus populáció számát a fentebbi fejezetben említett módon kiszámolva, tényleg optimális futást kapunk-e.

Gráf:
50 csúcsú teljes gráf

Ügynökök:
2 ügynök

Algoritmus:

Genetikus



Eredmény:

Az adatokat feldolgozva, a diagrammról leolvasható, hogy a Population size paramétert 0-tól 2000-ig futtatva, az eredmény lassan javulni látszik (kék vonal), és a számításhoz szükséges idő (sárga vonal) növekedni kezd. A zöld vonal mutatja az optimális populáció számát, és a piros vonal az optimális számmal történő futás idejét.

Látszik, hogy 800 körül helyezkedik el az optimális populáció mérete. 800 alatt gyorsabb a futás, viszont átlagosan nagyobb eredményt kapunk, 800 fölött pedig kicsivel, rosszabb eredménnyel, viszont jobb futási idővel szolgál a képlet szerint számoló megoldás.

Példa 5:

Cél:

Kis gráfok bejárása két ügynökkel. Mohó és genetikus algoritmusok eredményének összehasonlítása a legjobb megoldással.

A brute force algoritmus a legoptimálisabb megoldást adja eredményül, hiszen az összes lehetséges megoldás kipróbálja. Ehhez viszont rengeteg számolásra van szükség, ezért csak ilyen kis gráfra tud lefutni az algoritmus. Látható lesz, hogy egy 9 csúcsú gráf esetén is több mint 10000-szer több időbe telik a számolás.

Gráf:
5 és 9 csúcsú teljes gráfok

Ügynökök:
2 ügynök

Algoritmus:

Brute Force, Mohó és Genetikus

Eredmény				
Csúcsszám	Ügynökszám	Brute Force	Mohó	Genetikus
5	2	781,51	781,51	781,51
5	2	723,61	723,61	723,61
9	2	631,51	1073,44	670,97
9	2	615,69	647,72	665,58

Futási idő [ms]				
Csúcsszám	Ügynökszám	Brute Force	Mohó	Genetikus
5	2	8,2337	2,2450	2,9314
5	2	20,2608	2,5380	2,6602
9	2	243903,6850	2,7100	7,3507
9	2	250149,0205	3,0880	7,2972

Eredmény:

A kisebb gráfnál azt tapasztaljuk, hogy a mohó és a genetikus algoritmus is megtalálja az optimális megoldást, kicsivel kevesebb idő alatt.

A nagyobb gráfnál már nem ilyen pontosak az algoritmusaink. Az eredményhez relatíve közel járnak, de itt a futási idő már több mint 10000-szer kisebb lett.

Látható, hogy a mohó algoritmus futása nagyon eltér az optimálistól az egyik futás során, valószínű ott egy lokális minimumba ragadt.

Példa 6:

Cél:

Kis gráfok bejárása egy ügynökkel. A Christofides, a mohó és a genetikus algoritmusok eredményének összehasonlítása a legjobb megoldással.

Itt már tudunk kicsivel nagyobb csúcsszámú gráfokat is vizsgálni, hiszen a brute force algoritmust már nem terheli a több ügynökkel való számolás. Egy ügynökkel 13 csúcsú gráfot is futtatni lehet még.

Gráf:
10, 11 és 12 csúcsú teljes gráfok

Ügynökök:
1 ügynök

Algoritmus:

Brute Force, Christofides, Mohó és Genetikus

Eredmény					
Csúcsszám	Ügynökszám	Brute Force	Christofides	Mohó	Genetikus
10	1	1922,46	2255,00	2065,18	2001,74
10	1	1543,66	1818,26	1632,19	1593,75
11	1	1521,04	1869,61	1783,83	1622,58
12	1	1489,53	1710,34	1808,28	1660,84
12	1	1410,00	1833,57	1467,25	1413,42

Futási idő [ms]					
Csúcsszám	Ügynökszám	Brute Force	Christofides	Mohó	Genetikus
10	1	2219,7219	8,1030	2,9622	7,9960
10	1	2176,0682	8,7519	2,8047	7,9020
11	1	20419,9652	8,1937	3,2872	9,8188
12	1	221696,8035	12,9072	3,3622	11,6216
12	1	214284,3800	12,9557	3,1856	11,0244

Eredmény:

Jól látható, hogy szinte kivétel nélkül az eredmények szempontjából a legrosszabbtól a legjobbig a Christofides, a mohó a genetikus végül természetesen a brute force. Tudjuk, hogy a Christofides algoritmus csak egy Hamilton kört keres, nem számolja újra a dolgokat, míg a mohó és a genetikus újra és újra számol, kikerülve a lokális minimum eredményeket. Ez magyarázza azt, hogy a Christofides futása eredményezte a legrosszabbat.

A futási időnél is az elvárt eredményt kaptuk: legrosszabbtól a legjobbig, brute force, Christofides, genetikus és mohó. Látható hogy a nagyobb gráfokban a nagyobb komplexitás

miatt a brute force futási ideje rohamosan megugrik. A leggyorsabb a mohó algoritmus, de ez sem meglepő, hiszen gyorsan számol, de kevésbé pontosan, mint a genetikus. A Christofiges futási ideje nem nőtt annyira, mint a többi algoritmus, nagyobb gráfokra megelőzné a genetikust, és még a mohót is.