# 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

### Christofides

### 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

## Tesztkeretrendszer: