Szakdolgozat

UNIVERSITAS MISKOLCINENSIS

Miskolci Egyetem

Éttermi kiszállítás szimulációja és optimalizációja

Készítette:

Reisz Ákos

Programtervező informatikus

Témavezető:

Piller Imre

Miskolc, 2020

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Szakdolgozó Reisz Ákos (FZ3S16) mérnökinformatikus jelölt részére.

A szakdolgozat tárgyköre: optimalizáció, szimuláció, étterem, kiszállítás

A szakdolgozat címe: Éttermi kiszállítás szimulációja és optimalizációja

A feladat részletezése:

Az éttermi rendelések kiszállításánál megjelenő optimalizálási problémák szimulációja és vizsgálata. Az optimalizálás célja, hogy minél több megrendelést, minél rövidebb idő alatt, minél kisebb költséggel lehessen teljesíteni.

A dolgozat az egyszerűbb problémáktól az egyre valószerűbbek (így komplikáltabbak) felé haladva vizsgálja az optimalizálási problémák modelljét, lehetséges megoldási módjait. A legegyszerűbb esetnek az egy futár, egy étterem és egy kiszállítás esete tekinthető. Egy fokkal bonyolultabb változatban több kiszállítást is számításba kell venni, majd ezt követően a több étterem és több futár esetét is célszerű megvizsgálni.

Az optimalizálás során több célfüggvényt is érdemes megvizsgálni. Ilyen lehet például a kiszállítási idő vagy az egy kiszállításhoz tartozó megtett út minimalizálása.

A szimuláció és az optimalizálás Python programozási nyelv segítségével készül. Az algoritmusok vizsgálatához a különféle paraméterezések vizsgálata Jupyter munkafüzetekben történik. Az elkészült algoritmusok egy Python függvénykönyvtárba kerülnek. A működés helyességét az elkészített egységtesztek támasztják alá.

Témavezető: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje:

. . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Reisz Ákos**; Neptun-kód: `FZ3S16` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Mérnökinformatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Éttermi kiszállítás szimulációja és optimalizációja* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év .... .hó .... .nap

....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat .... szövegoldalt

.... program protokollt (listát, felhasználói leírást)

.... elektronikus adathordozót (részletezve)

....

.... egyéb mellékletet (részletezve)

....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: ....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: ....

a bíráló javaslata: ....

a szakdolgozat végleges eredménye: ....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Contents

Chapter 1

Bevezetés

Az éttermi rendelések hatékony kiszállítása elengedhetetlen manapság. Ezen hatékony kiszállítás egyik alappillére a megfelelő út kiválasztása. A vevő leadja a rendelést, majd minél hamarabb szeretné megkapni az étteremből rendelt csomagját. Az étterem érdeke, hogy minél kevesebb utat megtéve minél gyorsabban ki tudja szállítani a rendeléseket a kívánt helyre/helyekre. Egy jól megtervezett út pozitív hatással bír mind a két tényezőre. Egy hatásos útvonal által a vevő gyorsabban jut hozzá a rendeléséhez, ezáltal például a vevőt nem várakoztatják meg, praktikusan az étel is meleg és friss lesz még. Az étteremre vonatkozóan a pozitív vásárlói vélemények még több vásárlást ígérnek. Így tehát a jól megválasztott útvonalak rendkívűl fontosak. Mindezek mellett az étteremre más pozitív pénzügyi hatással is lehet, gondolok itt az üzemanyag kezdvező felhasználásától kezdve, a futárok jól megszervezett útjáig. Utobbi azért fontos, mert ezáltal több helyre ki tud szállítani egy adott futár, így több bevételt termel az adott étteremnek.

Az éttermi rendelések kiszállításánál megjelenő optimalizálási problémák igen széleskörűek. A mai modern technológia már lehetőséget ad ezen problémák megoldására, ezekre számos különféle megoldás született az évek alatt.

A szakdolgozatomban tárgyalt optimalizálandó problémákat a szerint lehet elkülöníteni, hogy mennyi az éttermek, futárok és a kiszállítások száma. Ezen varációk mindegyikét külön problémaként kezelem, külön eljárással dolgozom ki a megoldásukat.

Az eseteket külön vizsgálva mindegyik vizsgált esetben felírom a konkrét probléma megfogalmazását képpel illusztrálva.

Ezt követően rátérek a probléma megoldására. Dolgozatomban az optimalizálás célfüggvényét a megtett út hossza jelenti. A probléma bonyolultságát tekintve több algoritmust is felhasználtam az optimális útvonal meghatározásához. Ezen algoritmusok működését és a hozzájuk kapcsolódó fogalmakat alaposan részletezem a probléma megoldása közben.

Az esetek döntő többségéhez készült imlementáció is, ami a konkrét program kód fontosabb részeit tartalmazza magyarázattal. Ezen programkódokat megadott paraméterekkel tesztelve, képekkel illusztrálva látom be eredményességüket.

Chapter 2

A kiszállítási problémák általános modelljei

A kiszállítási probléma vizsgálatához először annak általános modelljét kell megadni. A vizsgált absztrakciós szintén a modell három alapvető eleme az étterem, a futár és a rendelés/kiszállítás helye (2.1. ábra).

Figure 2.1: Éttermi rendelés modelljének három fő tényezője, úgy mint az étterem, a futár és a rendelés helye

Az absztrak modellek nyilván nem írják le teljes részletességében a problémákat. Jelen esetben a következőket várjuk el a felírt modelltől. [?].

- Az éttermeket és a kiszállítások helyét pontszerűnek tekinti.
- Szóhasználatot illetően azt feltételezzük, hogy a kiszállítás egyetlen városon belül történik. (Ez a matematikai modell szempontjából nem lényegi megkötés, de a problémák leírását egyszerűsíti.)
- Nem feltételezi, hogy a futárnak lenne kapacitás, üzemanyag, vagy bármilyen hasonló jellegű limitációja.
- A kiszállítások bejárási sorrendjére vonatkozóan azon túlmenően, hogy mindegyiket be kell járnia a futárnak, külön nincsen. Az optimalizálási probléma eredményeként várjuk, hogy mi az optimális sorrend.
- A kiszállítás pontos idejére vonatkozóan nincs korlátozó tényező. A felvázolt modell az időt csak a megtett út függvényében képes tekinteni.
- Nem tekintünk a modell részének semmilyen közbeiktatott változást (például a rendelés lemondását) vagy egyéb problémát (például a futár járművének meghibásodását, kiszállítással kapcsolatos problémát).

Ezen egyszerűsítések mellett is láthatjuk majd, hogy igen változatos esetekben, komoly optimalizálási problémák megoldására lesz szükség.

2.1 A probléma modelljének lehetséges esetei

A szóbajöhető lehetőségek egyszerűbb áttekinthetősége érdekében adjunk meg hármasokat, melyekben az elemek az éttermek, futárok és kiszállítások számosságára vonatkoznak. A számosság itt lehet jelenthet egyet vagy többet. Előbbit 1-el, utóbbit pedig $*$-al fogjuk jelölni. Ezen jelölésrendszert használva összesen 8 lehetséges hármas adódna, úgy mint

$$(1,1,1),(1,1,*),(1,*,*),(1,*,1),(*,1,*),(*,*,1),(*,1,1),(*,*,*).$$

A következő szakaszban azt vizsgáljuk és indokoljuk meg, hogy melyek azok az esetek, amelyekkel ezek közül nem érdemes foglalkozni.

2.2 Figyelmen kívül hagyott és vizsgált esetek

- *Egy étterem, több futár, egy kiszállítás*: Nincs értelme vizsgálni, mivel egyetlen kiszállításhoz elegendő csak egy futár.
- *Több étterem, egy futár, egy kiszállítás*: Egy kiszállításnak szükségszerűen egy adott étteremből kell indulnia, tehát ez az eset nem értelmezhető.
- *Több étterem, több futár, egy kiszállítás*: Hasonlóan az előző esethez, nem tudjuk értelmezni, mert egy kiszállításhoz egyértelműen tartozik egyetlen étterem és egyetlen futár.

Az említett eseteket kihagyva az optimalizálási problémákat a következőkre tudjuk megadni:

- egy étterem, egy futár, egy kiszállítás,
- egy étterem, egy futár, több kiszállítás,
- több étterem egy futár, több kiszállítás,
- egy étterem, több futár, több kiszállítás,
- több étterem, több futár, több kiszállítás.

Az így adódó optimalizálási problémákat és azok lehetséges megoldásait a dolgozat a további fejezetekben mutatja be.

2.3 A város reprezentálása

Úgy tekintjük, hogy az éttermek és a kiszállítások helyei is egy városon belül vannak. A modell síkban gondolkozik, tehát egy étteremhez és egy kiszállítási helyhez is egy $(x, y) \in \mathbb{R}^2$ koordináta tartozik.

A közlekedési hálózat modellje egy gráf, amely azt adja meg, hogy melyik pontból melyikbe lehet eljutni. Egy kiszállítás útvonala tehát ezen pontok közötti szakaszok sorozatának tekinthető, melyen az éttermek és a kiszállítási helyek is egy-egy pontot jelentenek.

Chapter 3

Egy étterem, egy futár, egy kiszállítás esete

3.1 A probléma megfogalmazása

Ebben az esetben egy étterem található a városban, amelyből csak egy futár szállít ki mindig csak egy címre. Bonyolultságát tekintve a legrövidebb utat kell megtalálni az úthálózat gráfjának két pontja között, feltételezve azt, hogy egy pontból a másikba nem feltétlenül lehet közvetlenül eljutni, csak más pontok érintésével. Ezáltal egy gráf élein kell végigmenni és közben keresni a legrövidebb hosszúságú utat. Az optimális út meghatározása után venni kell ezen út hosszúnak kétszeresét, mivel a futár a szállítást követően vissza kell, hogy térjen az étterembe. A probléma szemléltetését fig:model1. ábrán láthatjuk.

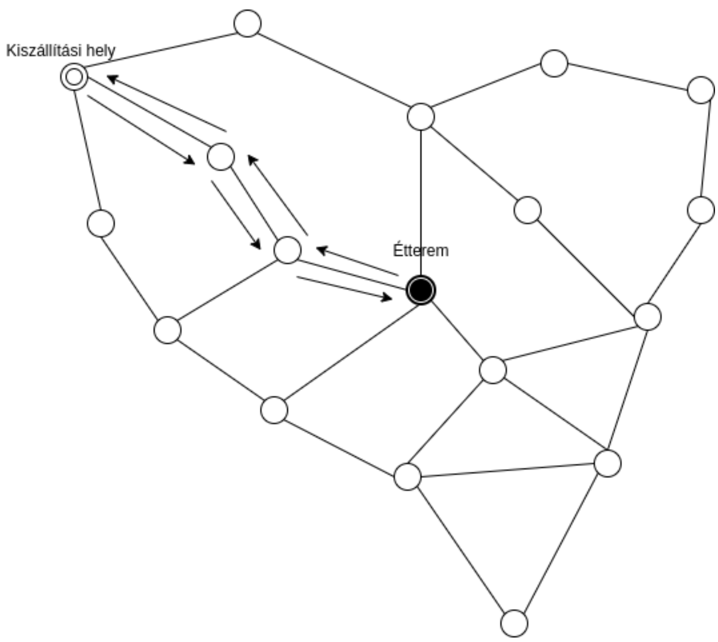

Figure 3.1: Egy étterem, egy futár, egy kiszállítás modellje

Két pont között a legrövidebb útvonal meghatározásához az A^* algoritmus egy igen hatékony megoldást ad. A következő szakaszban ennek részletezése következik.

3.2 A probléma megoldása

Az A^* algoritmusban a kiértékelő függvényünk egy heurisztikus függvény. Egy iteratív algoritmusról van szó. A fő ciklus minden iterációjánál az A^*-nak meg kell határoznia az általala kiterjesztendő utat. A szakasz költségét és a célhoz érésnek költségét veszi figyelembe, így az A^* meghatározza az $f(n) = g(n)+h(n)$ függvényt minimalizáló utat. Itt n-nel jelöljük a következő úton található csúcsot, ezáltal $g(n)$ a kezdőpontból n-ig tartó út költsége. A heurisztikus függvény $h(n)$-nel azonosítható, ez pedig nem más mint az n-től a célig vezető legkisebb költségű út költségének becslése.

Az algoritmus a meglátogatott pontokat tárolja, így nagy méretű problémák esetében nagy lehet a memóriaigénye is. Az esetünkben vizsgált problémák esetében ebből szerencsére nem jelentett problémát.

3.3 A megoldás implementálása

Érdemes az aktuálisan megoldandó feladat leírásához egy olyan formalizmust használni, amelyik egyszerűen adaptálható lehet a későbbiekben a valóságban is előforduló esetekhez. Abból indulhatunk ki, hogy a város úthálózatának megadását a térkép ismeretében néhány kézenfekvő átalakítási lépéssel jó lenne megoldani. Természetesen adódik, hogy így az úthálózat térképe egy kép legyen, amelyen be vannak jelölve azok az utak, amiken a futár haladhat. Mivel a kép lehet színes is, ezért egyúttal az éttermek és a kiszállítási helyek megkülönböztetésére is lehetőség adódik.

A térkép betöltéséhez az OpenCV függvénykönyvtárat használhatjuk. Ehhez az alábbi kódsor szükséges.

```python
import cv2
```

Definiáljunk egy `Node` nevű osztályt, amelynek inicializálása a következőképpen történik.

```python
def __init__(self, parent=None, position=None):
    self.parent = parent
    self.position = position

    self.g = 0
    self.h = 0
    self.f = 0
```

Ebben `g`-vel jelöljük a kezdőpontot, `h`-val a célt és `f`-el pedig a költséget.

A következő definiciók nélkülözhetetlenek a pontok összehasonlítási folyamatában.

```python
def __eq__(self, other):
    return self.position == other.position

def __hash__(self):
    return hash(self.position)
```

Az A^* algoritmus ezek segítségével a következőképpen írható le. Létrehozunk két csomópontot, úgy mint a `startNode` kezdő- és `endNode` végpontot.

```
startNode = Node(None, start)
startNode.g = startNode.h = startNode.f = 0
endNode = Node(None, end)
endNode.g = endNode.h = endNode.f = 0
```

Ezt követően inicializálunk két listát, melyek a már látogatott pontokat tartják majd nyilván (nyitott és zárt lista).

```
openList = []
closedList = set()
```

Megadjuk a kezdő csomópontot azáltal, hogy a nyitott listába rakjuk.

```
openList.append(startNode)
```

Ezt követezően ciklust indítunk, ami addig tart, amíg el nem éri a végpontot.

```
while len(openList) > 0:
```

A cikluson belül a következőképpen határozható meg a jelenlegi csomópont.

```
currentNode = openList[0]
currentIndex = 0
for index, item in enumerate(openList):
   if item.f < currentNode.f:
       currentNode = item
       currentIndex = index
```

A nyitott listából a zárt listába a következőképp rakjuk át az adott elemet:

```
openList.pop(currentIndex)
closedList.add(currentNode)
```

A célhoz vezető út meghatározása a következőképpen zajlik.

```
if currentNode == endNode:
    path = []
    current = currentNode
    while current is not None:
        path.append(current.position)
        current = current.parent
    return path[::-1]
```

Az út meghatározásában nagy szerepet játszik a navigáció; észak, dél, kelet és nyugat irányban tudunk elindulni, köztes irányok nincsennek. Meg kell hatázozni a csomópont pozicióját. Ezt követően egyértelművé kell hogy váljon, hogy közvetlenül el lehet-e érni azt. Le kell ellenőrizni, hogy az adott pont tényleg út-e. Mindezek után hozunk létre egy új csomópontot amit az után hozzáadunk az új lehetséges úthoz.

Mivel a probléma nagyon hasonló egy labirintusban való útvonal kereséséhez, ezért a térképet az algoritmus szempontjából `maze` változóként kezeljük.

```
newWay = []
for newPosition in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
```

```
    nodePosition = (currentNode.position[0] + newPosition[0],
                    currentNode.position[1] + newPosition[1])

    if nodePosition[0] > (len(maze) - 1) or
    nodePosition[0] < 0 or
    nodePosition[1] > (len(maze[len(maze)-1]) -1) or
    nodePosition[1] < 0:
        continue

    if maze[nodePosition[0]][nodePosition[1]] != 0:
        continue

    newNode = Node(currentNode, nodePosition)

    newWay.append(newNode)
```

Az algoritmus utolsó lépéseként megvizsgáljuk ezen lehetséges utakat, kiértékeljük a már fentebb említett `f`, `g` és `h` értékeket, majd ezek által kiválasztjuk a legkedvezőbbet.

```
for way in newWay:

    if way in closedList:
        continue

    way.g = currentNode.g + 1
    way.h = ((way.position[0] - endNode.position[0]) ** 2) +
            ((way.position[1] - endNode.position[1]) ** 2)
    way.f = way.g + way.h

    for openNode in openList:
        if way == openNode and way.g > openNode.g:
            continue

    openList.append(way)
```

Az implementációs rész elején említett `cv2` csomag a következők okok miatt szükséges. Adott egy kép, ami fekete $(0, 0, 0)$, fehér $(255, 255, 255)$, egy piros $(0, 0, 255)$ és egy kék $(255, 0, 0)$ színű pontokból tevődik össze (ahol a hármasokban az értékek a kék, zöld és piros csatornák intenzitását jelentik $[0, 255]$ egészes intervallumon). A fekete jelképezi a járhatatlan utat. Fehér színnel van jelölve a járható út. A piros szín mutatja meg a rendelési helyet, míg a kék az éttermet szimbolizálja. Az ilyen formában rendelkezésre álló kép feldolgozáshoz a következők inicializálások szükségesek.

```
img = cv2.imread("AStarProblem.bmp")
height, width, channels = img.shape

blue =  "[255   0   0]"
red =   "[  0   0 255]"
black = "[  0   0   0]"
white = "[255 255 255]"
```

```
mazeHelp = []
maze = []
```

A kép minden pontján áthaladva egy kettős ciklussal meghatározható a pixel pontos színe. Ebből egy tömböt kreálva elkészül az A^* algoritmushoz felhasználható térkép. Futtatva az algoritmust az optimális utat kiszínezi sárga színnel az étterem és a kiszállítás helye között.

```
for i in range(width):
    for j in range(height):
        if (str(img[i, j]) in white):
            mazeHelp.append(0)
        if (str(img[i, j]) in black):
            mazeHelp.append(1)
        if (str(img[i, j]) in red):
            mazeHelp.append(0)
            start = (i, j)
        if (str(img[i, j]) in blue):
            mazeHelp.append(0)
            end = (i, j)
    maze.append(mazeHelp)
    mazeHelp = []
path = aStar(maze, start, end)

for k in path:
    img[k] = [0, 255, 255]
cv2.imwrite('AStarResult.bmp', img)
```

3.4 A megoldás tesztelése

A megoldás teszteléséhez nélkülözhetetlen egy BMP kép. A program megfelelő futásához azt feltételezzük, hogy a kép hosszúsága és szélessége megegyezik. A fehér színnel kell kitelteni a járható utat, feketével a járhatatlant. Kék szín kell, hogy jelezze az éttermet, piros pedig a kiszállítási helyet. Amennyiben van lehetséges út az utóbbi két pont között, az alakalmazás megtalálja, valamint ha több út is tartozik hozzá, akkor a lehető legrövidebbet adja válaszul. Az eredmény egy képet generál, amin sárgával van feltüntetve az optimális út.

Egy kézzel rajzolt, várostérkép fig:model1problem. ábrán látható formában néz ki.

A megoldást felhasználva fig:model1result. ábrán látható képet kapjuk.

A kép kiválóan szemlélteti, hogy optimális utat adott eredményül, ezáltal az algoritmus felhasználható az adott feladathoz.

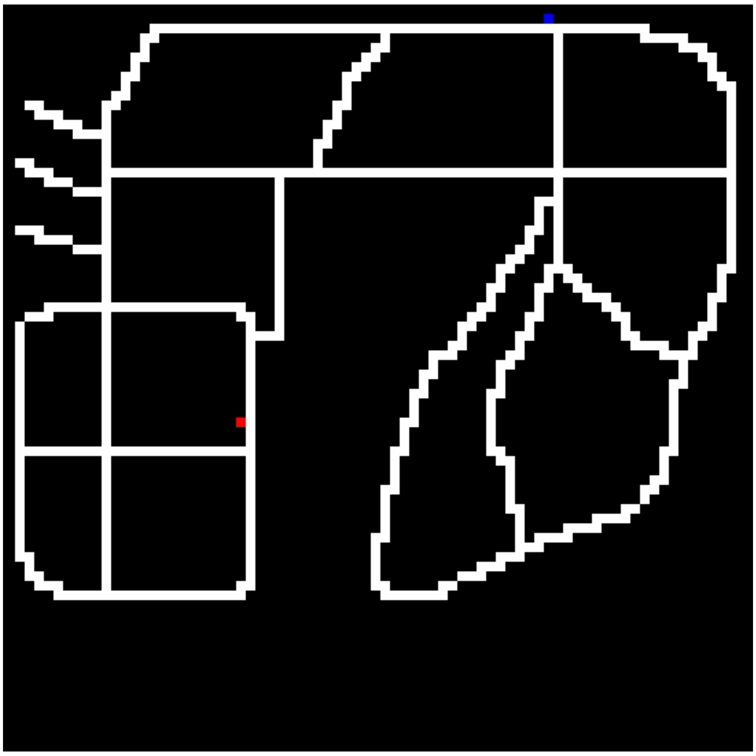

Figure 3.2: A teszteléshez használt térkép

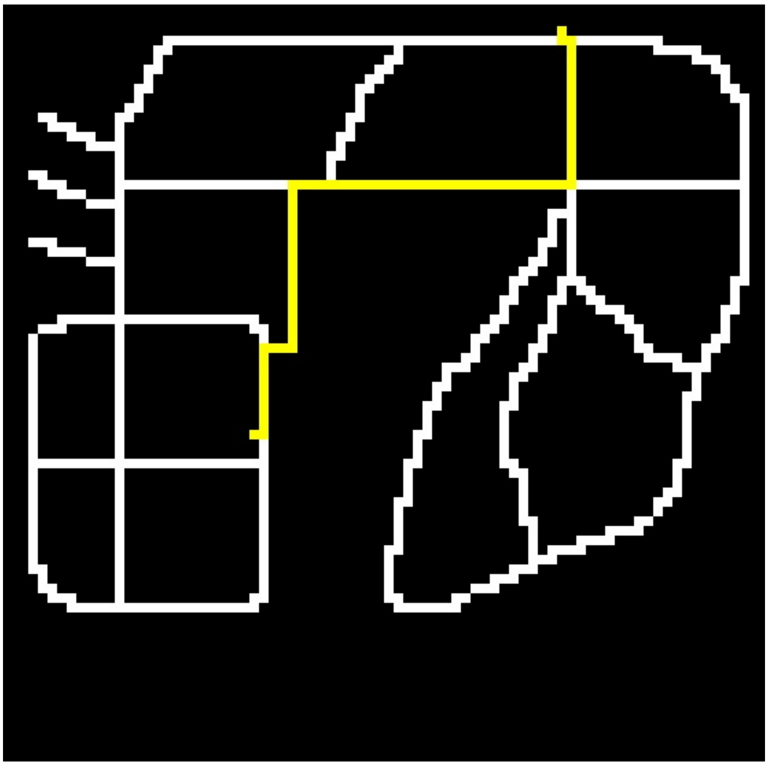

Figure 3.3: Az eredményül kapott útvonal

Chapter 4

Egy étterem, egy futár, több kiszállítás esete

4.1 A probléma megfogalmazása

Egy étterem, egy futár és több kiszállításnál az adott helyzet egészen visszavezethető a klasszikus utazó ügynök problémához. A futár elindul az étteremből, érinteni kell az összes kiszállítási pontot, valamint vissza kell érkeznie az étterembe, mindezt úgy, hogy a lehető legkisebb utat tegye meg. A probléma szemléltetése fig:model2. ábrán látható.

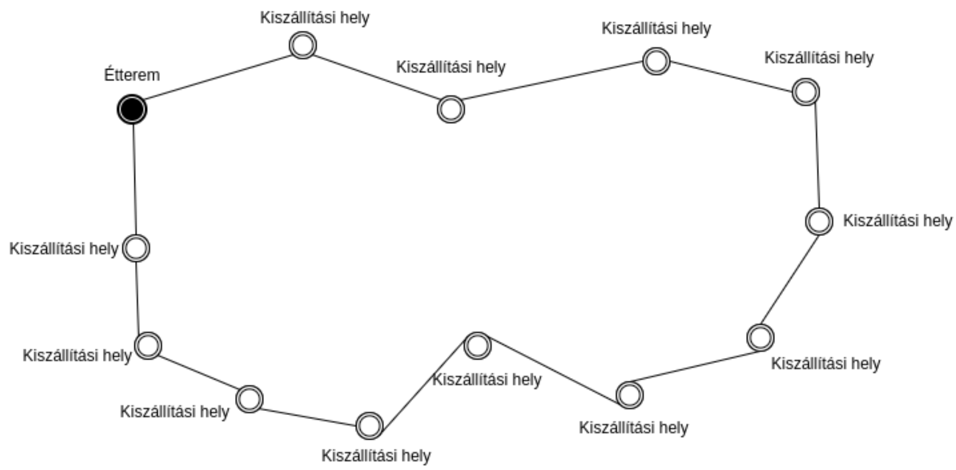

Figure 4.1: Egy étterem, egy futár, több kiszállítás modellje

4.2 A probléma megoldása

A probléma megoldásához egy nem determinisztikus módszer került implementálásra. Ebben egy úgynevezett *Gibbs-faktor* reprezentálja az új állapotba való áttérés valószínűségét.

A klasszikus utazó ügynök probléma matematikai megfogalmazása kiszállítási kritériumokra levetítve a következő. Az egy ügynökös utazó ügynök probléma esetén jelölje V a csúcsok (pontok) halmazát, $x_{i,j}$ azt, hogy az i. pontból megy-e közvetlenül út a j. pontba. Az $x_{i,j}$ értéke 1, ha útvonal köti össze a két pontot, különben 0:

$$x_{i,j} \in \{0,1\}, \quad i,j = 1,2,\ldots,n, i \neq j.$$

A $d_{i,j}$ jelöli az i. és a j. pont távolságát, n pedig a pontok számát. A célfüggvény az alábbi:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} d_{i,j} x_{i,j}.$$

A célfüggvénnyel magát a megtett távolságot szeretnénk optimalizálni. A pontba csak egy él fut be, tehát

$$\sum_{i=1}^{n} x_{i,j} = 1, \quad j = 1, 2, \ldots, n, i \neq j.$$

Valamint, minden pontból csakis egy él távozik, vagyis

$$\sum_{j=1}^{n} x_{i,j} = 1, \quad i = 1, 2, \ldots, n, i \neq j.$$

A sorrendiség a következő feltétel alapján érvényesül

$$u_i - u_j + n x_{i,j} \leq n - 1, \quad j = 2, 3, \ldots, n, i \neq j.$$

Itt u_i az i. pont, u_j a j. pont látogatási indexe, ahol az i. pontot hamarabb keresi fel az futár mint a j. pontot.

4.3 A megoldás implementálása

Összes lehetséges út száma $\dfrac{(n-1)!}{2}$. Ezek közül kell választanunk, ez ugyanis a Hamilton-körök száma az n pontból álló teljes gráfban ($n > 2$ esetén).

A Python implementációhoz az alábbi import szükséges:

```
from scipy.spatial import distance_matrix
```

A cél az, hogy listát készítsünk a pontokról, amelyek mindegyike két koordinátát tartalmaz (x, y) formában, amelyek 0 és 100 közötti véletlen egész számokként kerülnek kiválasztásra. Jelen esetben 10 ilyen pont lesz.

```
points = [random.sample(range(100), 2) for x in range(10)]
```

A pontok közötti távolságok kimutatását a program az alábbi módon oldja meg.

```
data = Points
points = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
df = pd.DataFrame(data, columns=['xcord', 'ycord'], index=points)
pd.DataFrame(
        distance_matrix(
                df.values,
                df.values
        ),
        index=df.index,
        columns=df.index
)
```

	1	2	3	4	5	6	7	8	9	10
1	0.000000	24.207437	13.892444	11.704700	52.000000	51.088159	27.802878	31.827661	60.926185	56.080300
2	24.207437	0.000000	37.483330	31.953091	75.591005	72.277244	36.674242	55.973208	84.534017	80.280757
3	13.892444	37.483330	0.000000	16.492423	41.773197	44.911023	35.440090	18.867962	50.447993	43.266615
4	11.704700	31.953091	16.492423	0.000000	44.102154	40.607881	18.973666	28.635642	53.000000	51.224994
5	52.000000	75.591005	41.773197	44.102154	0.000000	17.262677	52.201533	25.079872	8.944272	16.763055
6	51.088159	72.277244	44.911023	40.607881	17.262677	0.000000	42.201896	33.837849	21.587033	33.955854
7	27.802878	36.674242	35.440090	18.973666	52.201533	42.201896	0.000000	44.407207	60.307545	63.560994
8	31.827661	55.973208	18.867962	28.635642	25.079872	33.837849	44.407207	0.000000	33.060551	24.413111
9	60.926185	84.534017	50.447993	53.000000	8.944272	21.587033	60.307545	33.060551	0.000000	17.691806
10	56.080300	80.280757	43.266615	51.224994	16.763055	33.955854	63.560994	24.413111	17.691806	0.000000

Figure 4.2: Pontok közötti távolságmátrix

Ennek az eredménye fig:kimenet. ábrán látható.

Inicializáljuk a `pointCount` értékét 10-re, ugyanis ennyi helyre kell a futárnak eljutnia.

```
pointCount = 10
```

A `travel` egy adott számból álló lista (jelen esetben 10 számból áll), amely a pontok meglátogatására utal. Feltételezzük, hogy zárt hurokra van szükség, így az utolsó pont automatikusan csatlakozik az elsőhöz.

```
travel = random.sample(range(pointCount), pointCount);}
```

Elindítunk egy ciklust az adott értékekkel

```
for tlp in numpy.logspace(0, 5, num=100000)[::-1]:
```

Két pont véletlenszerű cseréjével új utat képzünk. Ezt úgy valósítom meg, hogy választok két számot az i-t és a j-t. Összeállítom a `newTravel`-t a régi travel másolásával az i indexig, majd összefűzöm a j-edik `travel`-t és egészen folytatom addig, amíg a j nem éri el az i-edik pontot, majd befejezem a `travel` többi részét.

```
[i, j] = sorted(random.sample(range(pointCount), 2));
newTravel = travel[:i] +
                         travel[j:j + 1] +
                         travel[i + 1:j] +
                         travel[i:i + 1] +
                         travel[j + 1:]
```

Bizonyos valószínűséggel a `travel` megkapja a `newTravel` értékét, az előzöekben említett csere miatt ez már változott. Az elképzelés az, hogy minimalizálni szeretnénk a pontok közti távolságok költségének összegét. Ehhez a Gibb-s faktort használtam fel, aminek lényege, az új állapotba való átmenet valószínűsége. Csak az i-edik és j-edik pontok közötti távolságokat szükséges összegezni, mivel a többi távolság ugyanaz a `travel`-ben, mint a `newTravel`-ben. Ha a faktor értéke nagyobb, mint 1, akkor az új költség alacsonyabb, a `travel` megkapja a `newTravel` értékét. Ez Python implementáció formájában a következőképpen néz ki.

```
traveld = sum([
        math.sqrt(
                sum([(
                        (
                          (points[travel[(k + 1) % pointCount]][d]) -
                          (points[travel[k % pointCount]][d])
                        ) **  2
                ) for d in [0, 1]])
        ) for k in [j, j - 1, i, i - 1]
])
newTraveld = sum([
        math.sqrt(
                sum([(
                        (
                          (points[newTravel[(k + 1) % pointCount]][d]) -
                          (points[newTravel[k % pointCount]][d]))
                        ** 2
                ) for d in [0, 1]])
        ) for k in [j, j - 1, i, i - 1]
])
if math.exp((traveld - newTraveld) / tlp) > random.random():
    travel = copy.copy(newTravel);
```

Az algoritmus végeztével már csak meg kell jeleníteni a kívánt pontokat, ez kirajzol egy gráfot, amely optimális utat ad. Ehhez a `matplotlib` függvénykönyvtárt használtam az alábbi módon.

```
plt.plot(
        [points[
                travel[i % pointCount]][0] for i in range(pointCount + 1)
        ],
        [points[
                travel[i % pointCount]][1] for i in range(pointCount + 1)
        ],
        'xb-'
)
plt.show()
```

4.4 A megoldás tesztelése

A `pointCount` állításával adhatjuk meg, hogy hány helyre is kell mennie a futárnak. Ennek módosításával több lefuttatott teszt után is megfigyelhető, hogy az összehasonlítások száma 65 ezer és 75 ezer között mozog. Egytől egyig az optimális utat adták. Mivel véletlenszerű számok összehasonlításán alapszik az algoritmus ezáltal az összehasonlítások száma igen magas, viszont stagnál bizonyos értékek között.

fig:tsp5location., 4.4., 4.5., 4.6., 4.7., 4.8., 4.9. és 4.10. ábrákon különböző számú pont esetén meghatározott optimális utakat láthatunk.

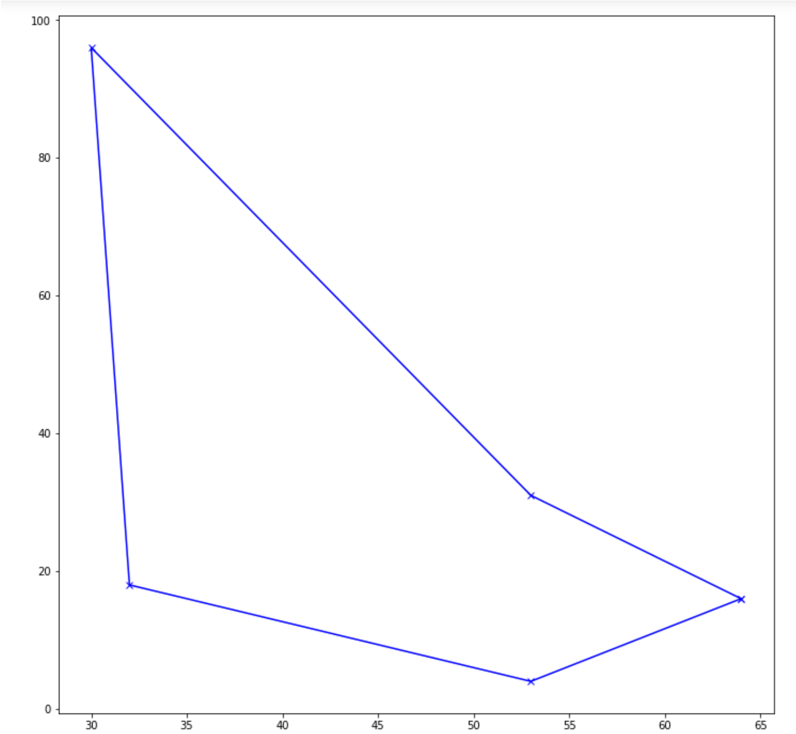

Figure 4.3: 5 kiszállítási hely esetén az összehasonlítások száma: 74 434

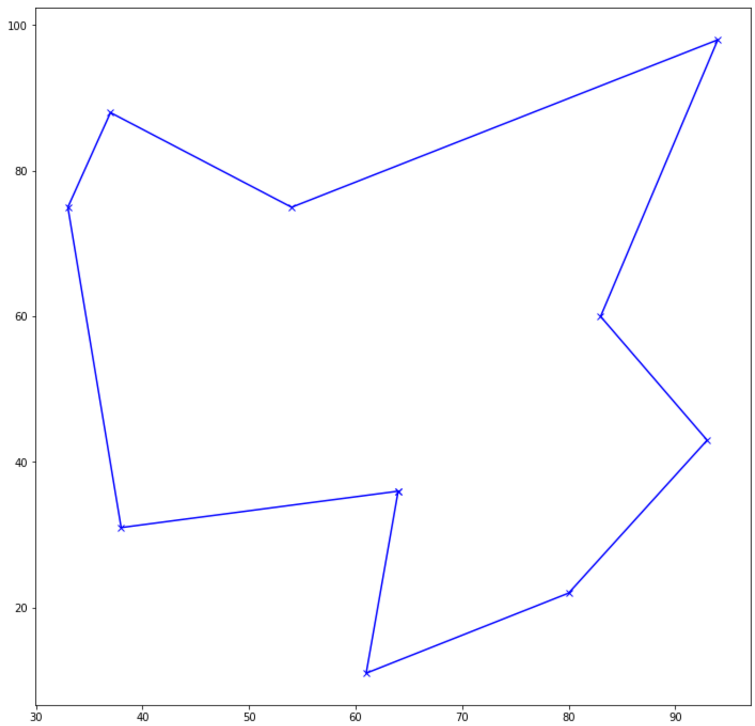

Figure 4.4: 10 kiszállítási hely esetén az összehasonlítások száma: 68 594

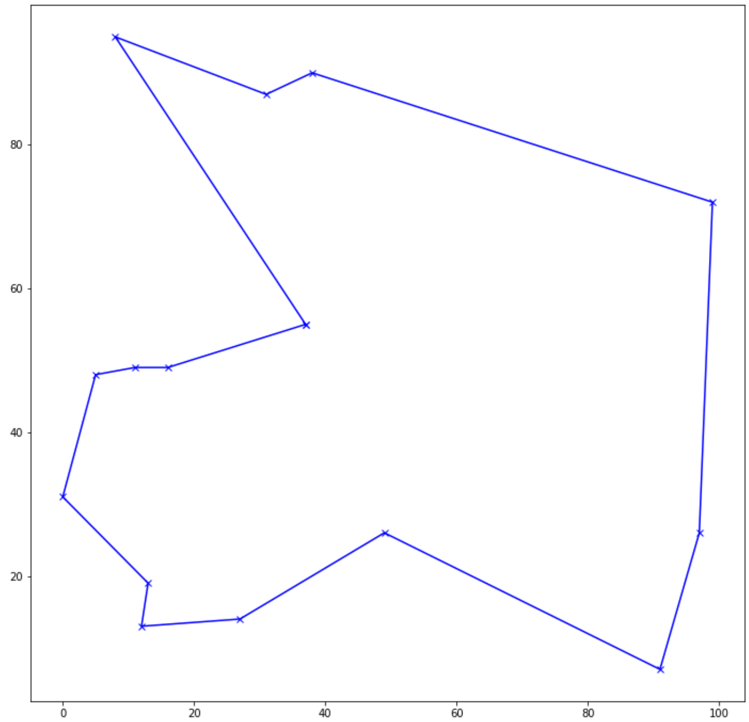

Figure 4.5: 15 kiszállítási hely esetén az összehasonlítások száma: 66 672

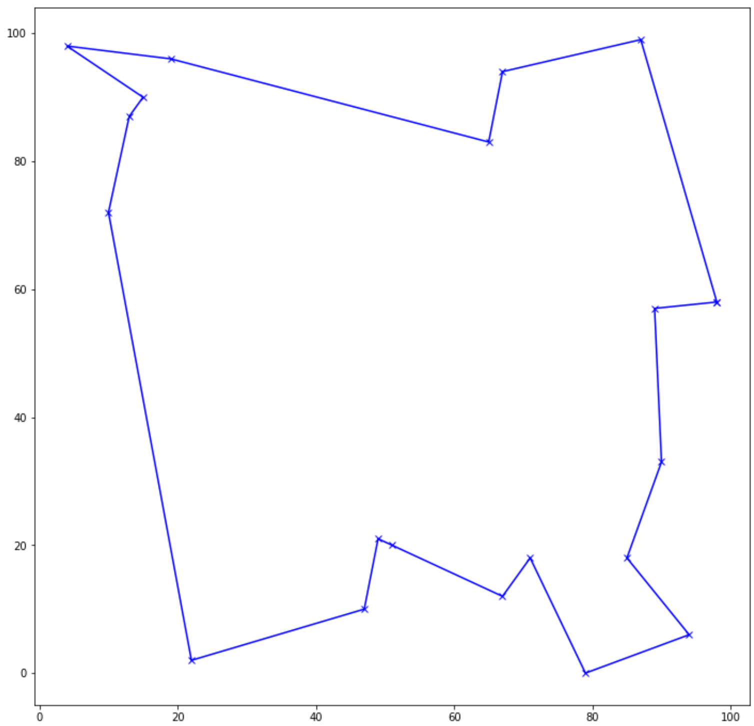

Figure 4.6: 20 kiszállítási hely esetén az összehasonlítások száma: 65 265

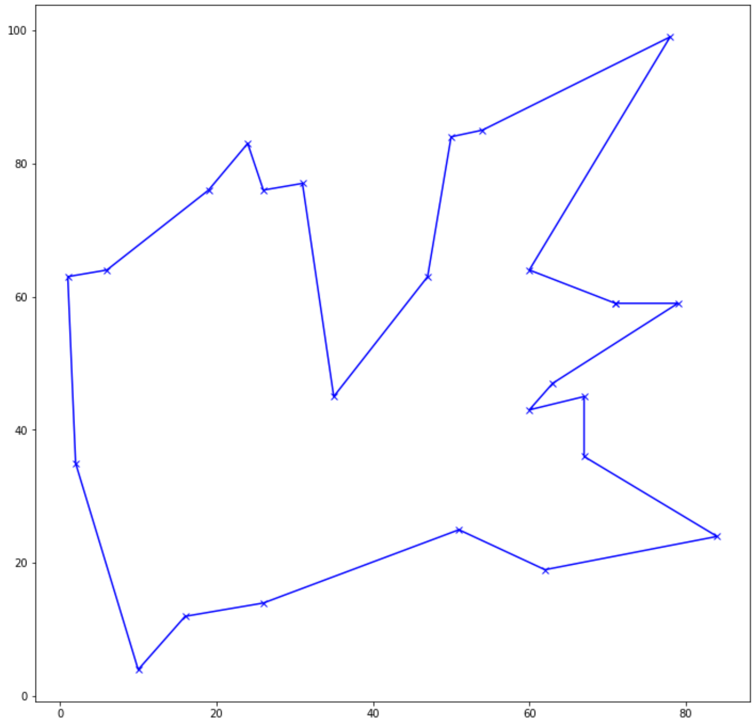

Figure 4.7: 25 kiszállítási hely esetén az összehasonlítások száma: 67 865

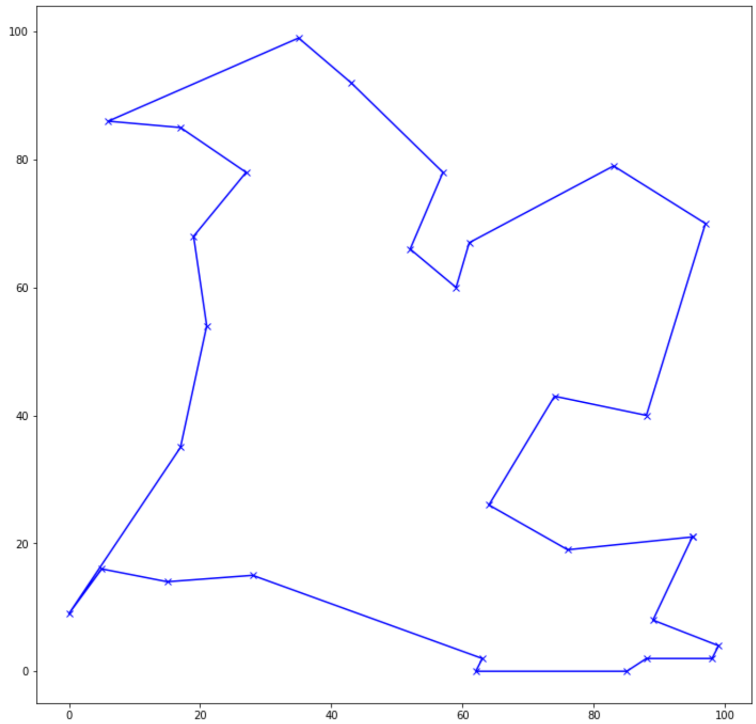

Figure 4.8: 30 kiszállítási hely esetén az összehasonlítások száma: 65 324

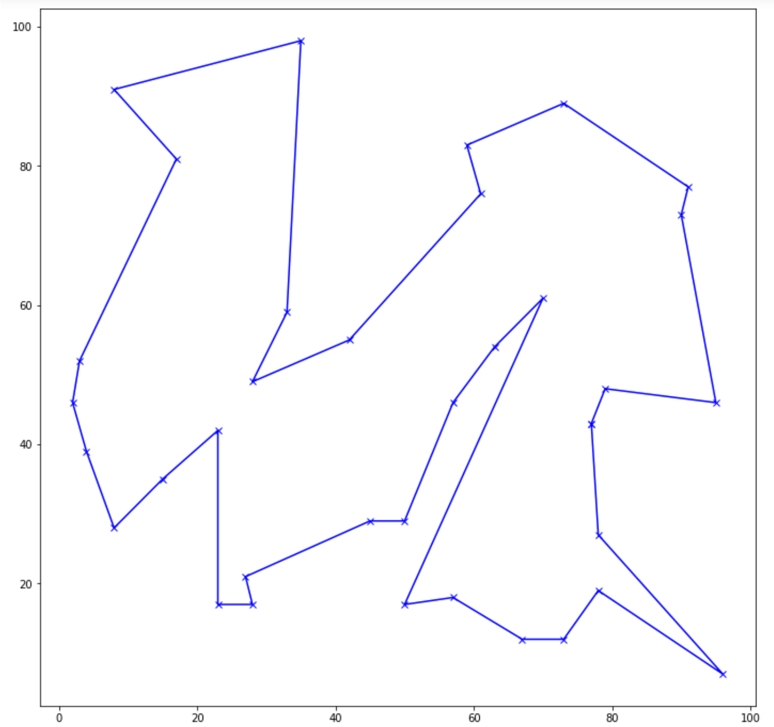

Figure 4.9: 35 kiszállítási hely esetén az összehasonlítások száma: 67 230

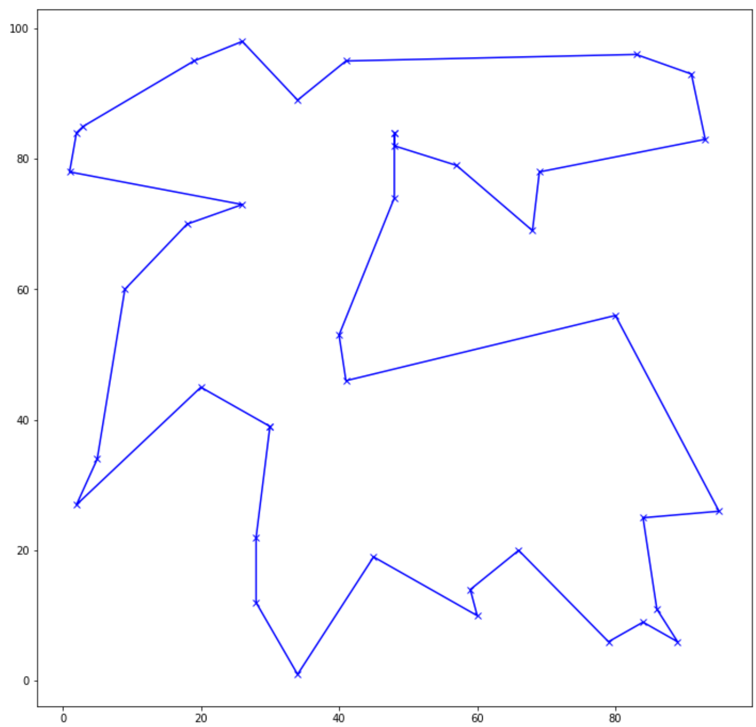

Figure 4.10: 40 kiszállítási hely esetén az összehasonlítások száma: 66 008

Chapter 5

Több étterem, egy futár, több kiszállítás esete

5.1 A probléma megfoglmazása

Több étterem esetén meg kell határozni néhány feltételt. Jelen helyzetben a futár elindul az egyik étteremből, kiszállít mindent, majd egy másik étterembe érkezik ezt követően, és annak a rendeléseit is kiszállítja. Időben is meg kell szabni néhány határt, miszerint az első étteremből való indulás pillanatáig beérkezett rendeléseket szállítja csak ki az összes étteremből. Miután sikeresen kivitte az összes rendelést visszatér a kezdő étterembe és kezdődik elölről a folyamat. Maga a folyamat egy önmagát ismétlő klasszikus utazó ügynök probléma, annyi eltéréssel, hogy nem ugyan abba az étterembe kell érkeznie ahonnan indult, hanem a hozzá legközelebb esőbe amelyikbe még nem járt az adott ciklusban. A probléma szemléltetése fig:model3. ábrán látható.

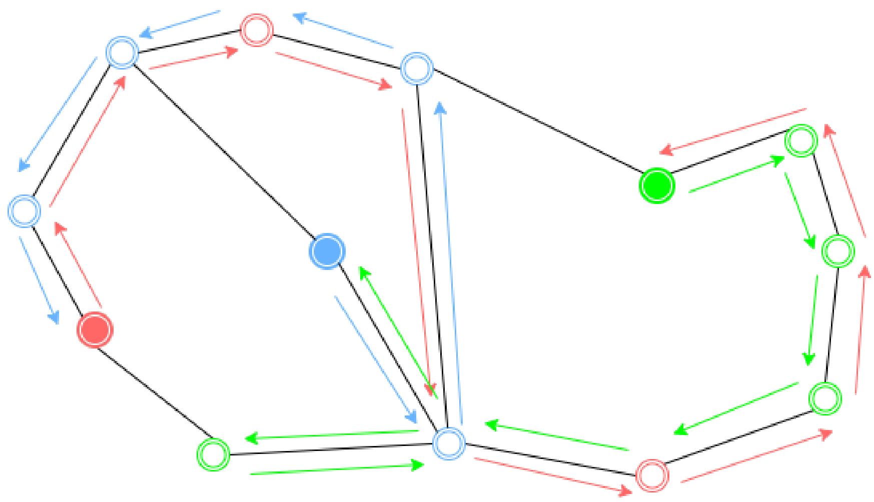

Figure 5.1: Több étterem, egy futár, több kiszállítás modellje

5.2 A probléma megoldása

Maga a probléma egy már megoldott helyzetre vezethető vissza. Az egy étterem, egy futár, több kiszállítás eseténél mindig ugyanabba az egy étterembe kellett visszatérnie a futárnak. Az ott alkalmazott módszerek érvényesek erre az esetre is, annyi eltéréssel, hogy ha kiszállította egy étterem rendeléseit az utolsó megállóhelye a folyamatban a következő étterem (nem az amiből indult). A folyamat következő lépése pedig innen indul, a futár kiszállítja az összes rendelést és egy harmadik étterembe érkezik utolsóképp. Ez mindaddig tart, amíg a futár meg nem látogatja az összes éttermet és vissza nem tér a kezdő, kiinduló étterembe.

5.3 A megoldás implementálása

Az előző megoldást felhasználva, pár változtatást el kellett végezni, hogy helyesen működjön erre az esetre.

Be kellett vezetni egy összegzést ami az éttermek számát határozza meg.

```
restaurantCount = 4
```

Ezt követően feltöltöttem egy listát véletlenszerűen generált x és y koordinátákkal. Maga a lista hossza az előbb meghatározott `restaurantCount`-al egyenlő.

```
restaurants = [random.sample(range(100),2)
               for x in range(restaurantCount)]
```

Szükségszerű volt bevezetni egy külső ciklust, ami az éttermek számáig megy. Ezáltal nyerhetőek vissza az adott étterem koordinátái.

```
for l in range(len(restaurants)):
```

A cikluson belül a pontok számát csökkentenünk kellett egyel, valamint az első helyre hozzáadni a `restaurants` adott `l` elemét. Ezáltal meg van oldva, hogy a ciklus mindig az éttermektől induljon.

```
travel = [random.sample(range(100), 2) for x in range(pointCount - 1)]
travel.insert(0, restaurants[l])
```

Meg kell még oldani, hogy a ciklus utolsó eleme a következő étterem koordinátája legyen. Figyelembe kellett venni azt is, hogy az utolsó étterem utolsó pontja az első étterem koordinátáival kell, hogy megegyezzenek. Szükségessé vált a kör megszakítása is a gráfban. Ezek a következőképpen néznek ki.

```
helperX = travel[i % pointCount]][0] for i in range(pointCount)
helperY = travel[i % pointCount]][1] for i in range(pointCount)

if l < (len(restaurants) - 1):
        helperX.append(restaurants[l+1][0])
        helperY.append(restaurants[l+1][1])
else:
        helperX.append(restaurants[0][0])
        helperX.append(restaurants[0][1])
```

5.4 A megoldás tesztelése

A módszer vizsgálatához 4 éttermet és 36 kiszállítási helyet vettem számításba. Az elkészített program által visszaadott útvonalak fig:tspMR1., 5.3., 5.4. és 5.5. ábrán láthatók. Jól megfigyelhető, hogy az utolsó étterem utolsó pontja visszatér az első étterem koordinátáihoz a (71, 42) pontba.

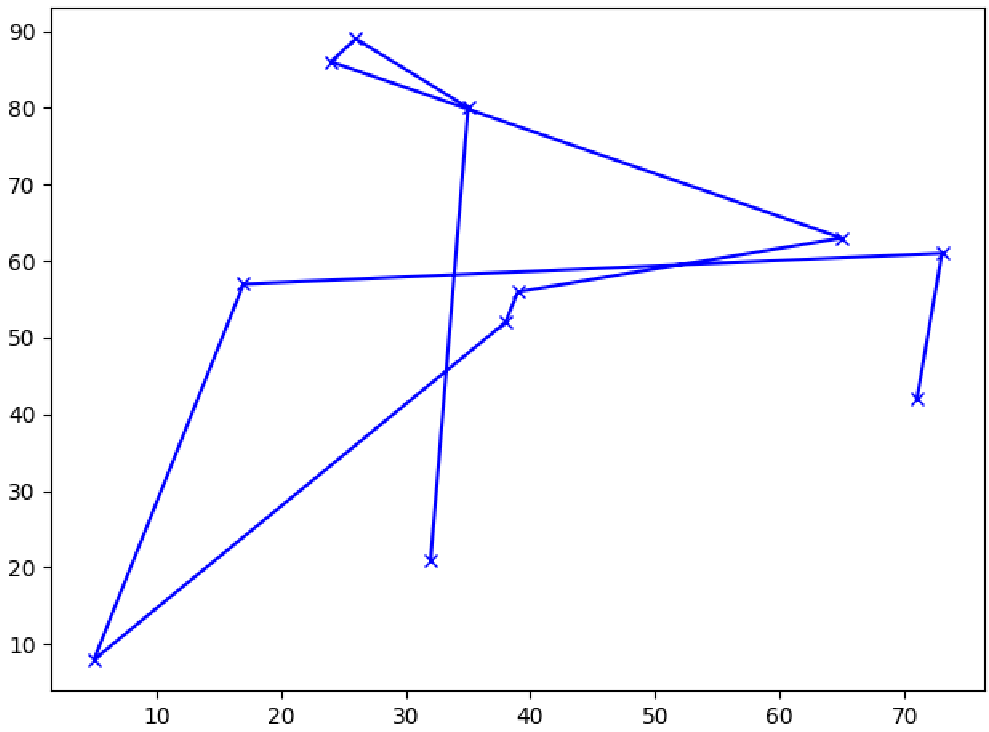

Figure 5.2: Első étterem pozicíója: (71, 42)

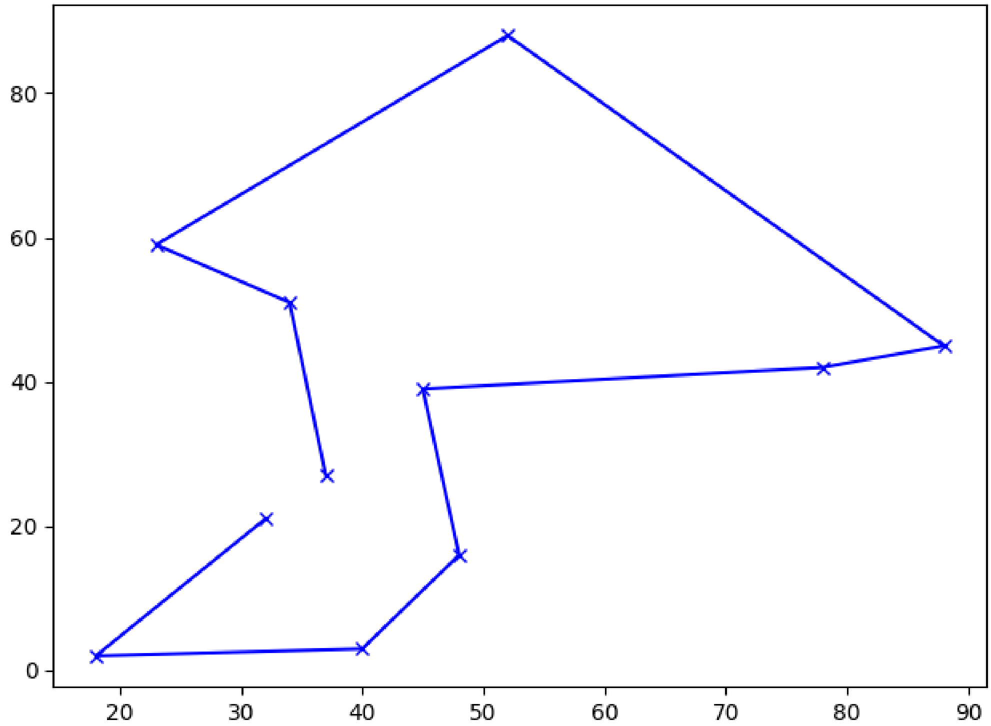

Figure 5.3: Második étterem pozициója: (32, 21)

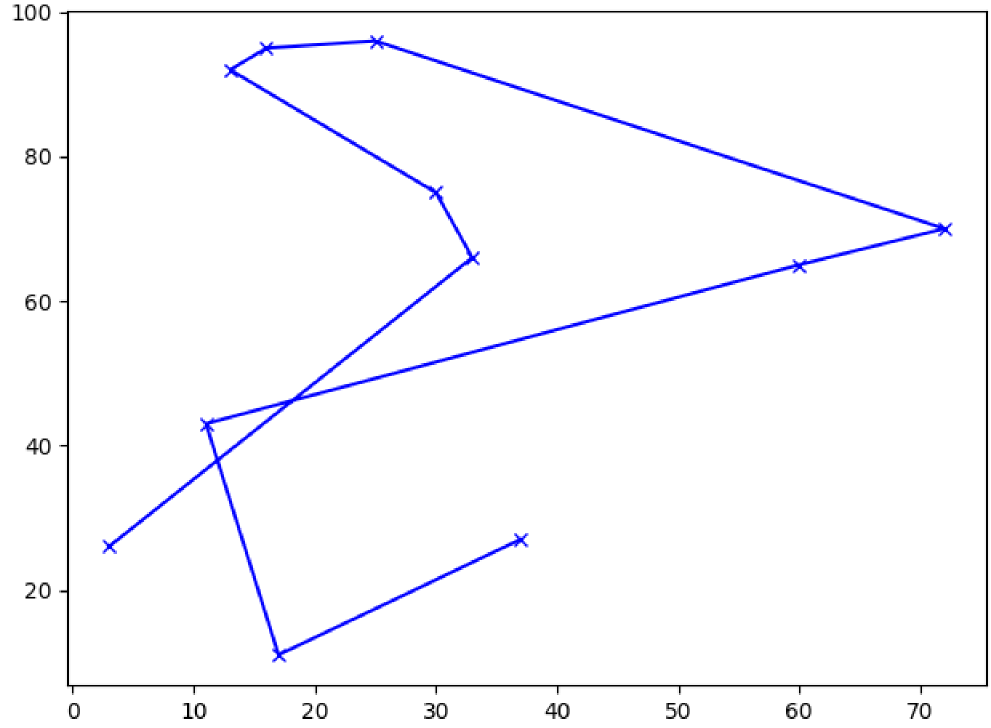

Figure 5.4: Harmadik étterem poziciója: (37, 27)

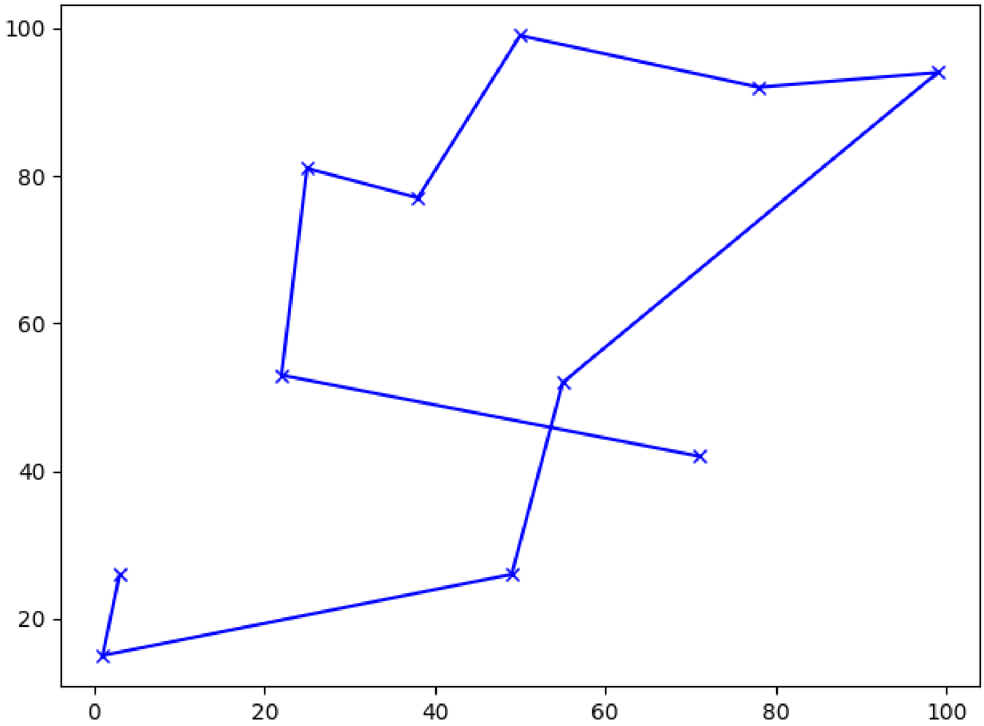

Figure 5.5: Negyedik étterem pozíciója: (3, 26)

Chapter 6

Egy étterem, több futár, több kiszállítás esete

6.1 A probléma megfoglmazása

Jelen eset reprezentálja az egy lerakatos több ügynökös utazó ügynök problémát. A feltételek meghatározásánál nélkülözhetetlen szempont, hogy határokat szabjunk az egyes futároknak, hogy ki milyen területre szállít ki. Ennek meghatározásánál fontos a kiszállítási címek közti táv figyelembe vétele. Ezek meghatározása után maga a probléma leegyszerűsíthető egy klasszikus utazó ügynök problémára. A probléma egy szemléltetését láthatjuk fig:model4. ábrán.

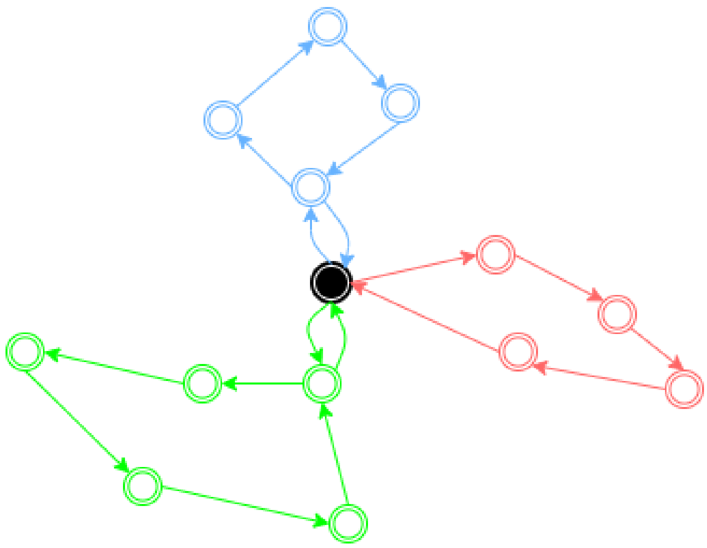

Figure 6.1: Egy étterem, több futár, több kiszállítás modellje

6.2 A probléma megoldása

A több ügynökös, egy lerakatos utazó ügynök probléma modellje kiszállítási helyzetekre szabva írható le.

A több ügynökös, egy lerakatos utazó ügynök probléma esetén legyen V a csúcsok halmaza, $x_{i,j}$ az, hogy megy-e az i. pontból út a j. pontba közvetlenül, $d_{i,j}$ az i. és j.

pont távolsága. Legyen m az ügynökök száma. Ezáltal a kapott célfüggvény:

$$\min \sum_{i=0}^{n} \sum_{j=0}^{n} d_{i,j} x_{i,j}.$$

Minden pontba csakis egy út indul, kivétel ez alól a 0. pont ami maga az étterem:

$$\sum_{i=1}^{n} x_{i,j} = 1, \quad j = 1, 2, \ldots, n, i \neq j.$$

Minden pontból csak egy út érkezik, kivétel ez alól a 0. pont ami maga az étterem:

$$\sum_{j=1}^{n} x_{i,j} = 1, \quad i = 1, 2, \ldots, n, i \neq j.$$

Az étteremre vonatkozó feltétel:

$$\sum_{i=1}^{n} x_{i,0} = m, \quad \sum_{j=1}^{n} x_{j,0} = m.$$

MTSP esetén a kövezkező megszorítások szükségesek:

$$u_i - u_j + p x_{i,j} \leq p - 1, \quad i, j = 1, 2, \ldots, n, i \neq j,$$

továbbá

$$\sum_{i \notin S} \sum_{j \notin S} x_{i,j} \geq r(S), \quad \forall S \subseteq V \setminus \{0\}, S \neq \emptyset,$$

ahol S a pontok egy részhalmaza, $r(S)$ pedig az, hogy ezt a részhalmazt minimum hány futárnak kell látogatnia. A definíció szumma része megmutatja, hogy minimum hány út megy be a vizsgált pontba. Az éttermet figyelmen kívűl hagyjuk m darab út hagyja el és m darab út megy ki belőle. Ezáltal a fenti egyenlőtlenség nem a futárok tényleges számát adná vissza.

Magát a problémát genetikus algoritmussal oldottam meg. A következő szakaszban ennek részletezésére kerül sor.

6.3 Megoldás genetikus algoritmus segítségével

A genetikus algoritmust számítógépes szimulációkkal reprezentálják. A keresési teret egyedek populációi alkotják. Ezen egyedeket lehet keresztezni (más szóval rekombinálni) és mutálni is, ezáltal új egyedek kreálhatóak. Fitnesz függvénynek nevezzük a keresési téren értelmezett célfüggvényt. Az algoritmus működése során új egyedeket képes létrehozni a rekombinációs és mutációs operátorokkal, valamint kiszűri a rosszabb fitnesz függvény értékkel rendelkező egyedeket, majd ezt követően kiszedi azokat a populációból.

Az algoritmus működésének a folyamata a következő lépésekben foglalható össze.

1. *Inicializáció*: Az induló populációt legkönnyebben véletlenszerű számokkal tudjuk létrehozni. Maga a populáció mérete erősen függ a problémától. A keresési térben az egyedek általában egyenletesen oszlanak el, viszont számos esetben több egyedet generál a feltételezhető optimum közelében.

2. *Kiválasztás*: Az összes eredményes generációban szelekcióra kerül az aktuális populáció egy része szaporodásra. A kiválasztás rendszerint fitnesz alapján történik, ahol a fitnesz függvény szerinti legfitebb egyedek valószínűbben kerülnek szelekcióra. Számos metódus az összes egyed fitneszét figyelembe veszi és ezek alapján keresi a legjobbat. Akadnak olyan metódusok is amik csak néhány véletlenszerűen kiválasztott példányt vizsgálnak. A példány minőségének mérésére használják a fitnesz függvényt. Ezen függvény bonyolultsága minden esetben a problémától függ.

3. *Szaporítás*: Egyoperandusú mutációval valamint kétoperandusú keresztezési műveletek által lehetséges a példányokból újabb példányokat kreálni. Ezen operátorokat véletlenszerűen használják.

4. *Leállás*: A genetikus algoritmus addig fut amíg be nem teljesül a leállási feltétel.

A módszer előnye: Kedvező eredménnyel használható majdnem minden problémára. Folytonos, illetve diszkrét problémáknál úgyszintén alkalmazható. A folyamatok könnyen páhuzamosíthatóak általa.

Hátránya: A megfelelő operátor kiválasztása nehéz. Paraméterezése és a leállási feltétel meghatározása sok időt igényel, valamint nem garantált a globális optimum elérése.

Elitizmus: Ebben az esetben a jelenlegi populáció legjobb egyedét mindig, módosítás nélkül visszük tovább az új populációba.

6.4 A megoldás implementálása

A genetikus algoritmus megvalósítása a `dustbin` modul `Dustbin` nevű osztálya, és a `galogic` modul `GA` nevű osztálya segítségével történt. Ezekhez hozzá tartoznak még a `main`, `globals`, `population`, `route` és a `routeManager` nevű modulok. A következőkben ezek bemutatására kerül sor.

6.4.1 A `Dustbin` osztály

A `Dustbin` osztály reprezentálja a bejárható pontokat az útvonalban, annak x és y koordinátájával (6.2. ábra).

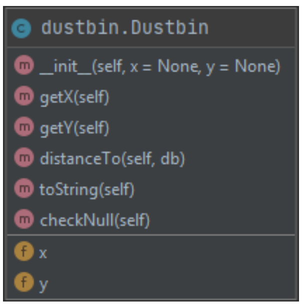

Figure 6.2: Dustbin osztály metódusai és adattagjai

Inicializáló metóduson kívűl található benne még olyan folyamat ami felel, az x és y értékek feldolgozásával. Ezen felül az euklidészi távolság számítás is itt foglal helyet. Továbbá található még benne egy formázó metódus, ami a szöveget helyesen formázottan adja vissza.

6.4.2 A `galogical` modul

Az eljárás lényegi, logikai része a `GA` osztályban valósul meg (6.3. ábra).

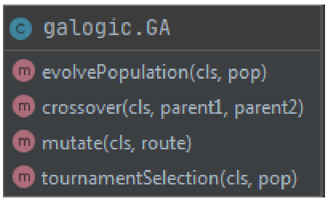

Figure 6.3: Galogical osztály metódusai

Az `evolvePopulation` a populáció fejlesztésére használt metódus. Tartalmaz egy ellenőrzést, hogy kívánunk-e elitizmussal élni a futás alatt vagy sem. Ezenfelül a keresztezés és mutáció operátorokat meghívó programkódok is itt foglalnak helyet.

```python
@classmethod
def evolvePopulation(cls, pop):

    newPopulation = Population(pop.populationSize, False)

    elitismOffset = 0
    if elitism:
        newPopulation.saveRoute(0, pop.getFittest())
        elitismOffset = 1

    for i in range(elitismOffset, newPopulation.populationSize):
        parent1 = cls.tournamentSelection(pop)
        parent2 = cls.tournamentSelection(pop)
        child = cls.crossover(parent1, parent2)
        newPopulation.saveRoute(i, child)

    for i in range(elitismOffset, newPopulation.populationSize):
        cls.mutate(newPopulation.getRoute(i))

    return newPopulation
```

A keresztezési operátor implementációját a következő metódussal oldottam meg.

```python
@classmethod
def crossover(cls, parent1, parent2):
    child = Route()
    child.base.append(Dustbin(-1, -1))
    startPos = 0
    endPos = 0
    while startPos >= endPos:
        startPos = random.randint(1, numNodes - 1)
```

```
        endPos = random.randint(1, numNodes — 1)

    parent1.base = [parent1.route[0][0]]
    parent2.base = [parent2.route[0][0]]

    for i in range(numDeliverer):
        for j in range(1, parent1.routeLengths[i]):
            parent1.base.append(parent1.route[i][j])

    for i in range(numDeliverer):
        for j in range(1, parent2.routeLengths[i]):
            parent2.base.append(parent2.route[i][j])

    for i in range(1, numNodes):
        if i > startPos and i < endPos:
            child.base[i] = parent1.base[i]

    for i in range(numNodes):
        if not(child.containsDustbin(parent2.base[i])):
            for i1 in range(numNodes):
                if child.base[i1].checkNull():
                    child.base[i1] =  parent2.base[i]
                    break

    k = 0
    child.base.pop(0)
    for i in range(numDeliverer):
        child.route[i].append(RouteManager.getDustbin(0))
        for j in range(child.routeLengths[i]—1):
            child.route[i].append(child.base[k])
            k += 1
    return child
```

A mutáció operátor kódja az alábbi.

```
@classmethod
def mutate (cls, route):
    index1 = 0
    index2 = 0
    while index1 == index2:
        index1 = random.randint(0, numDeliverer — 1)
        index2 = random.randint(0, numDeliverer — 1)

    route1startPos = 0
    route1lastPos = 0
    while route1startPos >= route1lastPos or route1startPos == 1:
        route1startPos =
          random.randint(1, route.routeLengths[index1] — 1)
        route1lastPos =
```

```
          random.randint(1, route.routeLengths[index1] - 1)

    route2startPos = 0
    route2lastPos = 0
    while route2startPos >= route2lastPos or route2startPos == 1:
        route2startPos =
          random.randint(1, route.routeLengths[index2] - 1)
        route2lastPos= random.randint(1, route.routeLengths[index2] - 1)

    swap1 = []
    swap2 = []

    if random.randrange(1) < mutationRate:
        for i in range(route1startPos, route1lastPos + 1):
            swap1.append(route.route[index1].pop(route1startPos))

        for i in range(route2startPos, route2lastPos + 1):
            swap2.append(route.route[index2].pop(route2startPos))

        del1 = (route1lastPos - route1startPos + 1)
        del2 = (route2lastPos - route2startPos + 1)

        route.route[index1][route1startPos:route1startPos] = swap2
        route.route[index2][route2startPos:route2startPos] = swap1

        route.routeLengths[index1] = len(route.route[index1])
        route.routeLengths[index2] = len(route.route[index2])
```

A `tournamentSelection` metódus kiválasztja a legfitebb kromószómahalmazt.

```
@classmethod
def tournamentSelection (cls, pop):
    tournament = Population(tournamentSize, False)

    for i in range(tournamentSize):
        randomInt = random.randint(0, pop.populationSize-1)
        tournament.saveRoute(i, pop.getRoute(randomInt))

    fittest = tournament.getFittest()
    return fittest
```

6.4.3 Az algoritmus globális változói

Az algoritmushoz tartozó globális változók egy `globals` nevű modulba kerültek. Az inicializáló adatokat foglalja magába, ezen kívűl még két függvényt is tartalmaz. Az inicializáló adatokat három részre lehet osztani.

1. A koordináták tartományát lehet vele manipulálni.

```
xMax = 100
```

```
yMax = 100
seedValue = 1
numNodes = 40
numGenerations = 20
```

2. A populációra vonatkozó adatok itt módosíthatóak.

```
populationSize = 20
mutationRate = 0.02
tournamentSize = 1
elitism = True
```

3. Ezen részben lehet megadni a futárok számát is.

```
numDeliverer = 2
```

A `random_range` és a `route_length` függvények véletlenszerű adatok generálására szolgálnak.

6.4.4 A `main` modul

Maga a futtatható metódus. Ebben értékelődik ki az optimális útvonal valamint egy diagram ami y tengelyén látható a fittnesz függvény érték (távolság), x tengelyén pedig a generációk száma.

6.4.5 A `population` modul

Az egyedeken végzett műveletekhez szükséges oszály (6.4. ábra).

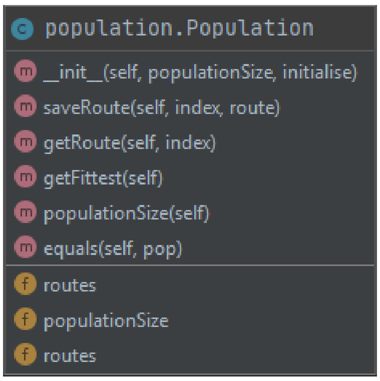

Figure 6.4: Population osztály elemei

Tartalmaz egy inicializáló metódust, ezen felül itt kapott helyet az út mentése, kiolvasása is. A legfitebb érték meghatározásán kívűl a populációk összehasonlításával is foglalkozik az oszály.

6.4.6 A `Route` osztály

Az optimális út kiszámításához használt osztály (6.5. ábra). Az alapvető inicializáláson kívűl jelen vannak még a helyes út kiszámításához használatos metódosok. Végezetül egy szövegformázó metódus is itt kapott helyet.

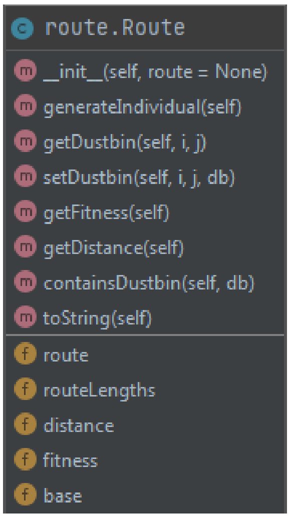

Figure 6.5: A `Route` osztály elemei

6.4.7 A `Routemanager` osztály

Az oszály egyedeket kezel, három osztálymetódusból áll amik az optimális út eléréséhez nélkülözhetetlenek (6.6. ábra).

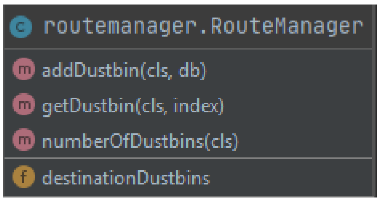

Figure 6.6: A `Routemanager` osztály elemei

6.5 A megoldás tesztelése

A genetikus algoritmus működését megvizsgáltam elitizmus használatával és a nélkül. Ezek eredményeit a következő szakaszok foglalják össze.

6.5.1 Elitizmus nélkül

Az algoritmus paraméterei a következők voltak:

- A kiszállítási helyek száma: 40
- Generációk száma: 40
- Populáció nagysága: 20
- Elitizmus: Hamis
- Futárok száma: 3

A futtatás eredményei fig:MTSPMultiDepo1. és 6.8. ábrán láthatók.

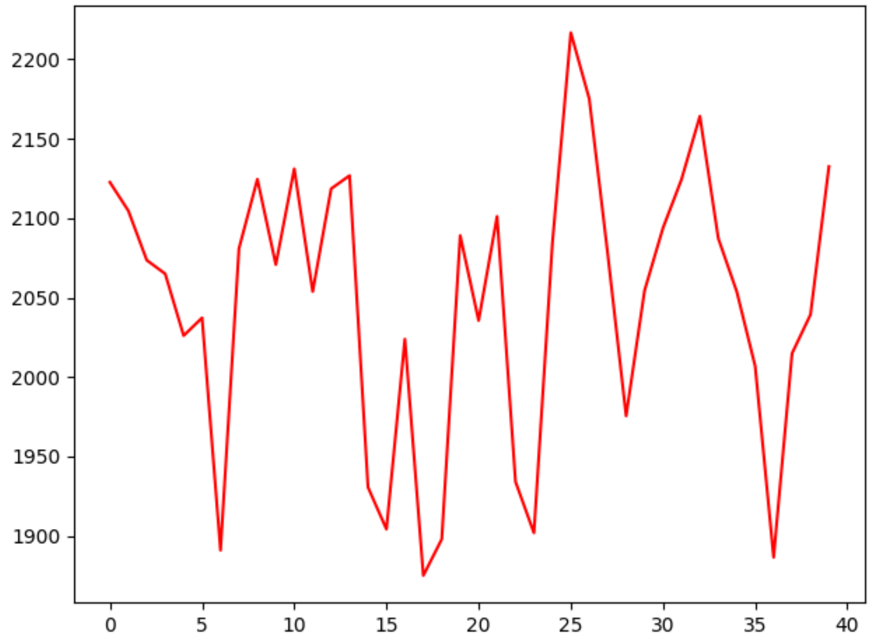

Figure 6.7: Fitnessz értékek generiónként elitizmus nélkül

```
Final Route:
|(90,52)|(76,59)|(74,33)|(1,74)|(17,16)|(11,80)|(20,84)|(59,97)|(53,59)|(96,26)|(27,90)|(29,96)|(76,61)|(99,10)|(96,15)|(88,61)|
(90,52)|(72,14)|(34,45)|(4,95)|(6,18)|(5,4)|(79,78)|(84,71)|(91,68)|
(90,52)|(93,4)|(32,90)|(28,74)|(68,42)|(24,80)|(80,68)|(76,96)|(98,73)|(89,29)|(26,5)|(67,89)|(96,61)|(78,30)|(77,24)|(60,19)|(32,75)|
```

Figure 6.8: Futárok útjai elitizmus nélkül

6.5.2 Elitizmussal

Az előző paraméterezéssel, viszont elitizmus alkalmazásával fig:MTSPMultiDepo2. és 6.10. ábrákon látható eredményeket adta az algoritmus.

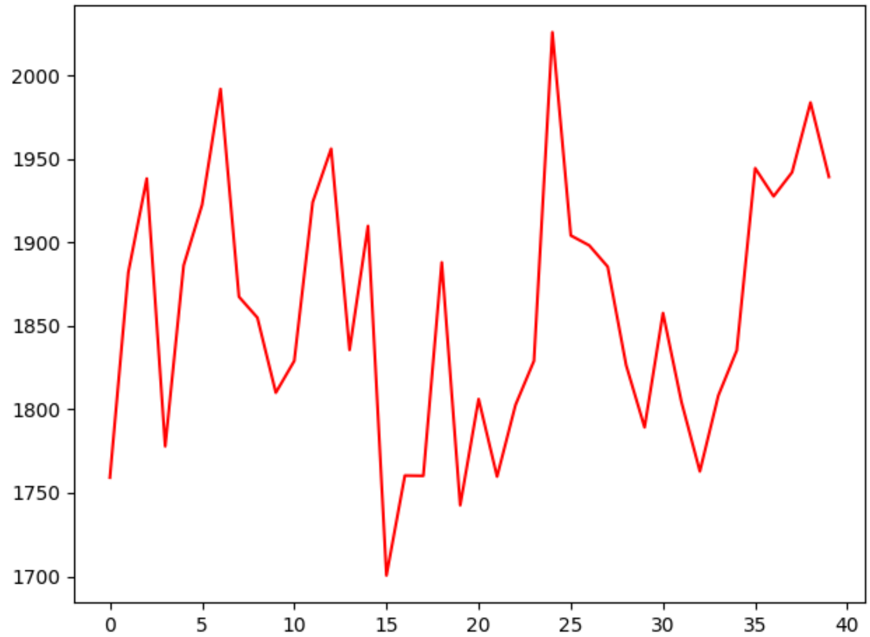

Figure 6.9: Fitnessz értékek generiónként elitizmussal

```
Final Route:
|(7,40)|(74,18)|(78,35)|(67,21)|(82,73)|(97,75)|(40,45)|(44,39)|(76,72)|
(7,40)|(11,9)|(6,6)|(5,60)|(34,23)|(67,28)|(98,9)|(86,40)|(74,39)|(76,61)|(17,79)|(47,72)|(36,57)|(79,72)|(21,52)|
(7,40)|(77,53)|(88,31)|(6,18)|(51,60)|(51,57)|(90,90)|(50,28)|(82,36)|(100,17)|(52,37)|(15,81)|(89,10)|(34,19)|(48,29)|(65,13)|(23,7)|(12,42)|
```

Figure 6.10: Futárok útjai elitizmussal

Chapter 7

Több étterem, több futár, több kiszállítás esete

7.1 A probléma megfoglmazása

Ez az eset tisztán leírja a több lerakatos több ügynökös utazó ügynök problémát. Jelen esetben tisztázni kell, hogy a futárok különböző éttermekből indulnak ki. Próbálnak keresni egy olyan utat, aminek költésége viszonylag kicsi, tehát a közeli helyekhez tartozó csomagot veszik csak fel és szállítják ki. Ezt követően a legközelebbi kiszállítási helyet vizsgálják. Figyelembe veszik a szükségesen meglátogatni kívánt étterem körüli kiszállítási helyket, és ez alapján választják meg az útjukat. Amennyiben ez az út túl költésges lenne, akkor próbálnak keresni egy másikat célt, aminél kevesebb út befektetésével több címet tud meglátogatni. Mindezek mellett, hogy a folyamatban ne legyen többszöri meglátogatás, a futároknak tudniuk kell, hogy ki hol járt már, hol történt meg a kiszállítás sikeresen. A model egy szemléltetését láthatjuk fig:model5. ábrán.

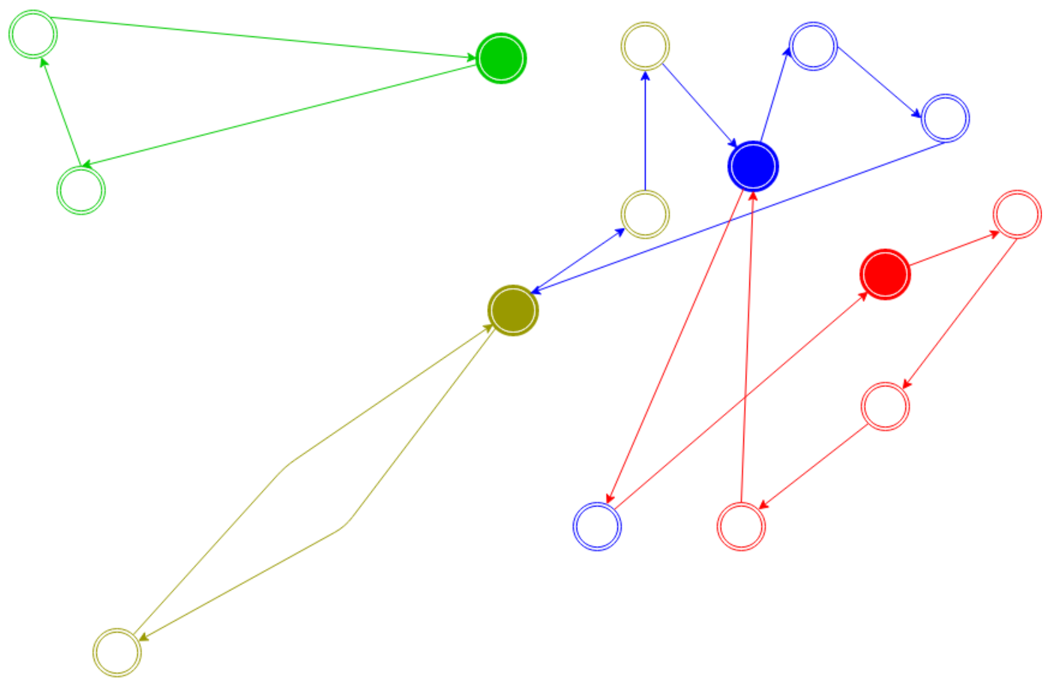

Figure 7.1: Több étterem, több futár, több kiszállítás modellje

7.2 A probléma egy lehetséges megoldása

Első lépésként véletlenszerű pontokat kell generálnunk. Ezek jelképezik majd a kiszállítási helyeket. Ezt követően meg kell határoznunk az éttermek számát, majd legenerálni őket véletlenszerű helyekre. Később meg kell határoznunk egy átlagos távolságot, ami a különböző kiszállítási pontok közötti távolságok átlagának felel meg. Nyilván kell tartanunk, hogy ki melyik étteremből rendelt, hogy ezáltal mindenképp az éttermet kelljen előbb meglátogatni, ha a csomag még egyik futárnál sincs. Tekintettel kell lenni a futárok aktuális helyezetére, hiszen ha elfogyott a csomagja, keresni fogja a legközelebbi pontot ahova viheti majd a szállítmányt. Nélkülözhetetlen meghatározni a futárok útjait. Ehhez szükséges az előzőekben meghatározott átlag szakasz. A folyamat felépítésére tekintettel szükséges egy fő ciklus, mely a futárok számáig megy. Ezen belül megy egy másik ami a futár útjain megy végig. Ezen megoldás gyakorlati alkalmazása esetén lehetséges optimalizálni a kiszállításokat egy ilyen összetett modell esetén is.

Ezen optimalizálási feladat részletesebb vizsgálata még további kutatások tárgyát képezi.

Chapter 8

Összefoglalás

A dolgozatom témája az éttermi rendelések optimalizációja és szimulációja, amely segít meghatározni, az optimális útvonalat a futároknak kiszállítás közben. Az ezekhez készült saját fejlesztésű szoftvereket elkészítése és működése került bemutatásra különböző paraméterezésekkel. A példákban szereplő szemléltető diagramok direkten az adott paraméterezett szituációhoz lettek generálva.

A szoftverek tökéletesre fejlesztése hosszú időt vehet igénybe. Társítani lehetne egy navigációs szoftverhez, aminél nem véletlenszerűen generált pontok lennének, hanem a feltérképezné az utakat és a szerint alakítaná ki a pontokat. További fejlesztési lehetőségként két verziót tudnék elképzelni; egy központi rendszert, amely az éttermekben lenne, és egy kliens alkalmazást, amely a futároknál. Az étteremben kiválasztanák, hogy milyen esetről van szó, milyen paraméterekkel, és az autómatikusan kiosztaná azt a kliensek között. A futároknál lévő mutatná vizuálisan a futár aktuális helyét és a helyes utat is, amin haladva a legkisebb távot teszi meg.

Ezen fejlesztéseket eszközölve az éttermek a folyamat során megfelelően tudják kihasználni a futárjaikat. Ezáltal a vevői elégedettség mellett időre és pénzügyi előnyökre tehetnek szert.

Summary

The topic of my thesis is the optimization and simulation of the delivery of restaurant orders, which helps to determine the optimal route for couriers during delivery. I presented the operation of self-developed software for situations with parameterized data. The illustrative diagrams in the examples were generated directly for the given situations. The development of a perfect software can take a long time in this scenario.

In a further development, the developed algorithms could be associated with a navigation software that would not use randomly generated points, instead it would map the routes and map the points accordingly. It would have two versions, a server one that would be in restaurants and a client that would be at couriers. In the restaurant, they would choose what case it is and with what parameters, and it would automatically allocate it among the client versions. A pointer at the couriers would visually show the current location of the courier on the map. In addition, the shortest route to the destination would be visible.

By making these improvements, the restaurants need to be able to take advantage of their couriers in the right way. This allows them to gain time and financial benefits in addition to customer satisfaction.

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).