



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Tóth Márk Andor

ALGORITMUSOK

OPTIMALIZÁLÁSA GPU-N

KONZULENS

Dr. Szegletes Luca

BUDAPEST, 2022

Tartalomjegyzék

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
1.1 Fejezetek	9
2 Elméleti háttér	11
2.1 Jármű útvonaltervezési probléma	11
2.1.1 Korlátozott kapacitású járművek	12
2.2 Utazóügynök probléma	13
2.3 Hangyakolónia algoritmus	14
2.3.1 Alternatívák	15
2.4 Adatreprezentáció	15
2.4.1 Szomszédsági mátrix	16
2.4.2 Compressed Row Storage formátum	16
3 Technológiai háttér	19
3.1 Nyelv és párhuzamos technológia	19
3.2 Véletlenszám generálás.....	20
3.2.1 Pszeudorandom generátorok.....	21
3.2.2 Véletlenszámgenerátorok vizsgálata.....	21
3.2.3 CuRAND	22
3.3 GPGPU programozás.....	23
3.3.1 Optimalizálás memória sávszélességre.....	23
3.3.2 Optimalizálás utasításvégrehajtás-kihasználásra	24
3.3.3 Memória limitáció.....	25
3.3.4 Kernel indítás.....	25
4 Megvalósítás	27
4.1 Hangyakolónia algoritmus	27
4.1.1 Egyszálú megvalósítás	29
4.1.2 Többszálú megvalósítás	30
4.1.3 Valószínűségek kiszámítása az élekhez.....	32
4.1.4 Útvonalak generálása	34
4.1.5 Legjobb útvonal kiválasztása.....	36

4.1.6 Feromonértékek frissítése	38
4.2 Rulettkerék algoritmus.....	38
4.2.1 Rulettkerék algoritmus a gyakorlatban	38
4.3 Legjobb útvonal meghatározása	40
4.3.1 Determinisztikus módszer.....	40
4.3.2 Probabilisztikus módszer	41
4.4 Szálanként több hangya szimulálása.....	42
4.5 Algoritmus paraméterek	43
4.5.1 Technikai paraméterek.....	43
4.5.2 Hangolható paraméterek	44
5 Eredmények.....	46
5.1 Adatkészlet.....	46
5.2 A mérések módja	47
5.3 Kiértékelés	49
5.3.1 CPU és GPU összehasonlítása.....	49
5.3.2 BlockCount és BlockSize	50
5.3.3 Iterációk száma	51
5.3.4 Kernelek időaránya	52
5.3.5 Heurisztikus algoritmus pontossága	53
6 Összefoglalás.....	54
7 Továbbfejlesztési lehetőségek	55
8 Irodalomjegyzék.....	56

HALLGATÓI NYILATKOZAT

Alulírott Tóth Márk Andor, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 12. 08.

.....
Tóth Márk Andor

Összefoglaló

Az utazóügynök probléma és a jármű útvonaltervezés olyan problémák, amelyre sok feladat, például a fuvarozó cégek üzemanyag megtakarítása, CNC gépek optimális működése vagy egyes munkaütemezési problémák visszavezethetők.

Az utazóügynök probléma során megadott állomásokat kell bejárni a lehető legkisebb költséggel, míg a jármű útvonaltervezési probléma esetében az állomásokat több jármű használatával kell lefedni.

Mivel az alaproblémák optimális megoldásának megtalálására – bizonyos speciális eseteket leszámítva – még nem létezik polinomiális időben elvégezhető algoritmus, ezért a közelítő algoritmusok, például a hangyakolónia optimalizáció kiemelt szerepet játszik ezen problémák gyakorlati kezelésében.

A hangyakolónia típusú algoritmusokat a hangyák természetben megfigyelt élelemszerző viselkedése inspirálta. Használatukkal sok olyan feladat is megoldható nagy pontosságú közelítéssel, amelyekre a pontos megoldást meghatározó algoritmus még nem ismert, vagy nem is létezik. Sikerrel alkalmazták őket az útvonaloptimalizációs problémákon túl például a számítógépes képfeldolgozásban éldetektlásra, valamint klasszifikációs feladatokhoz.

Ezen algoritmusoknak a jövőbeli felhasználhatóság szempontjából fontos tulajdonsága, hogy párhuzamos futtatásra is alkalmasak. A dolgozatomban megvizsgálom ennek egy lehetséges megvalósítását a CUDA platformon.

Az algoritmus időigényét összehasonlítom az egyszálú megvalósítással, hogy kiderüljön mekkora gyorsulás várható a párhuzamos technológia alkalmazásával. A kapott eredmények támpontként szolgálhatnak más problémák vizsgálatához, mivel a hangyakolónia algoritmusok az említett problémákon túl optimalizációs feladatok széles palettájára adaptálhatók.

Abstract

The traveling salesman and vehicle routing problems provide solution to a wide range of real-world tasks, such as optimization of fuel consumption in logistics, CNC toolpath generation or specific job-shop scheduling problems.

The traveling salesman problem requires finding the shortest path to visit a given number of vertices with a single agent. The vehicle routing problem on the other hand requires such optimization using multiple agents (vehicles).

There is no known exact algorithm that runs in polynomial time to solve any of these problems except some special cases. Thus, approximation methods such as ant colony optimization algorithms are needed to handle them in practical applications.

These algorithms can give accurate approximations for tasks without a known (or existing) exact solution method. Ant colony optimization were used to successfully solve a number of different problems, such as digital image processing edge detection or classification tasks.

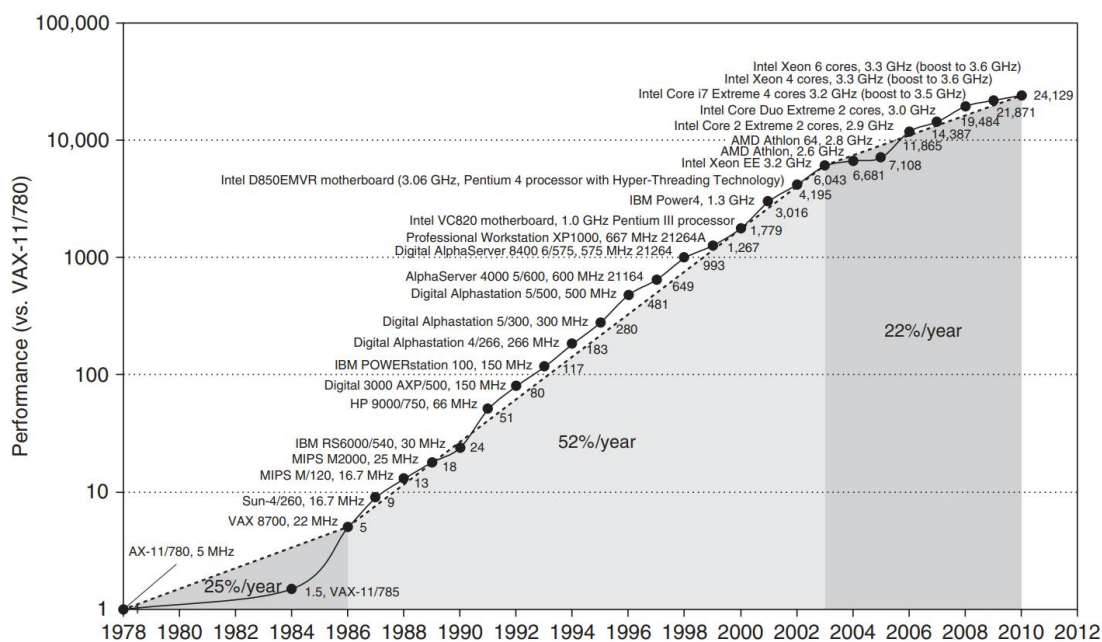
Ant colony optimizations can be ran in a massively parallel manner which seems to be an important aspect of future algorithm design. In my thesis, I show an example implementation using the CUDA parallel computing platform.

I compare the time requirements of this with the original, single-threaded approach. The results can be later used to construct algorithms to solve different problems using ant colony optimization, because it is a powerful tool when handling a wide variety of optimization tasks.

1 Bevezetés

A számítástechnológia fejlődése hosszú időn át az Intel egymérnöke, Gordon E. Moore által megfogalmazott megfigyelésnek megfelelően alakult. Moore törvénye azt mondja ki, hogy a hardvereszközökben építhető tranzisztorok száma, ezzel párhuzamosan azok számítási teljesítménye két évente megduplázódik.

A megfigyelés évtizedeken át kiállta az idő próbáját, azonban a gyártástechnológia fejlődése elérte azt a határt, ami felett a félvezető chippek tovább nem zsugoríthatók össze. A távolságok már atomokban mérhetők, az órajelek nem növelhetők tovább, a keletkező hő elvezetése is problémát jelent. Ennek köszönhetően a teljesítménynövekedés lassulni kezdett.



1. ábra: Processzorok teljesítménynövekedése a VAX 11/780 számítógéphez viszonyítva [1]

Az egyszálú utasításvégrehajítás ezen korlátai miatt a figyelem hamar alternatív megoldásokra terelődött. A fejlődés fenntartására több másfajta megközelítés is szóba jöhet. Napjainkban intenzíven kutatott terület a kvantumszámítógépek előállítása. Ezekkel a számítógépekkel elméletben olyan problémák is megoldhatóvá válnak majd, amelyekre a mostani hardverek nem képesek, ilyen például az egész számok prímfelbontása kvantum Fourier-transzformáció segítségével [2]. Ugyan kvantum hardvereket már egyes vállalatok képesek gyártani, de használatuk a hétköznapiakban még nem terjedt el, az átlagember számára nehezen hozzáférhetők.

Ugyanakkor létezik egy hardver, amely igen elterjedt az asztali és mobil eszközökben, továbbá a szuperszámítógépekben, és lehetővé teszi a leggyorsabb CPU-knál is nagyobb számítási teljesítmény elérését. Ez az eszköz a grafikus hardver, másnéven videokártya, amely párhuzamos architektúrájának köszönhetően sok algoritmus futási sebességét a többszörösére képes növelni. Használata ugyanakkor eltérő algoritmizálási szemléletet is követel.

Bizonyos műveleteket könnyedén lehet párhuzamosítani, míg másokat egyáltalán nem. Sokszor ennek eldöntése nem egyszerű feladat és önmagában nem csak az algoritmustól, de az adatoktól is függhet.

Vannak olyan módszerek azonban, amikről már a magas szintű felépítésük alapján sejteni lehet, hogy a parallelizációval akár több nagyságrenddel is csökkenthető a futási idejük.

Egy ilyen algoritmuscsalád a hangyakolónia optimalizáció (ant colony optimization, ACO) is, amellyel hatékonyan közelíthetők egyes optimalizációs problémák legjobb megoldásai. A valódi legjobb megoldást ugyan általában nem képesek megadni elfogadható idő alatt, azonban közel-optimális eredményeket igen. Ez sok alkalmazásnál elég is, mert a pontos megoldás megtalálása a bemenet méretének függvényében akár exponenciális időt is igényelhet. Az ACO algoritmusokat használják többek között hálózatok tervezéséhez (energia, távközlési), pénzügyi előrejelzésekhez, és a logisztikában.

A dolgozatomban ezt az alkalmazást, azon belül is az utazóügynökök (travelling salesman problem, TSP) és a jármű útvonaltervezési problémák (vehicle routing problem, VRP) megoldását fogom megvizsgálni.

Mivel ezek a problémák bizonyítottan NP-nehezek, gyakorlati alkalmazásokban az optimális útvonal megkeresése nem alkalmazható stratégia. A hangyakolónia algoritmus előnyei azonban a párhuzamos számítási hardverek terjedésével egyre gyorsabb és pontosabb közelítéseket tesz lehetővé.

Mindkét probléma megoldására bemutatok egy-egy egyszerű, egyszerű futtatásra szánt algoritmust, majd kifejtem az ezekben rejlő párhuzamosítási lehetőségeket és azok futásidőre gyakorolt hatását.

A dolgozatom kiindulási pontjaként Katona Géza értekezése [3]szolgált, amelyben a két pont közötti számítógépes útvonaltervezés ACO segítségével történő

párhuzamosítása köré épül. Habár az alapproblémák, amelyekre megoldást kerestünk eltérnek, a feladatok között mégis van hasonlóság. Az általam bemutatott algoritmus átalakítható úgy, hogy használható legyen ilyen útkeresésre is. Az értekezésben érintett egyik probléma a gyakorlati alkalmazások során előforduló feladatok nagy tárigénye volt, így ennek kiküszöbölésének kiemelt figyelmet szenteltem.

A megvalósításhoz C++ nyelvet használtam, mivel hardverközeli módon lehet benne leírni a programokat. A parallelizációt az Nvidia CUDA platformjára építve végeztem el. Ennek előnye, hogy sok dokumentáció és fejlesztést segítő szoftvertermék érhető el hozzá online, valamint a használatához szükséges eszközök megfizethetőek az átlagember számára is. A rendelkezésre álló források egy része nem könyv vagy publikáció formában érhető el, hanem például az Nvidia blogján. Azokat a bejegyzéseket, amelyek a cég saját mérnökei publikáltak, megbízható forrásnak tekintem és hivatkozni fogok rájuk.

1.1 Fejezetek

A dolgozatom több nagyobb logikai egységre tagolható:

A Bevezetésben összefoglalom miről fog szólni a dolgozat, valamint ismertetem a felépítését. Ezen túlmenően arról is ejtek néhány szót, hogy miért pont ezeket a problémákat választottam.

Az Elméleti háttér fejezetben mutatom be a TSP és VRP problémákat. A problémák modellezésére gráfokat használok, így itt a gráfelméletben elterjedt jelöléseket használom.

Bemutatok továbbá más, elterjedten használt megközelítéseket is, amelyekkel kezelhetők a fenti problémák. Röviden ismertetem ezek előnyeit, illetve hátrányait.

Leírom továbbá az ACO általános működését. Az algoritmizálás módját és nehézségeit, valamint a párhuzamosítás menetét nem itt, hanem a Megvalósítás fejezetben fogom részletezni.

Emellett bemutatok egy módszert az adatok tárolására, amely ritka gráfok reprezentációja során különösen hasznos a grafikus hardver memória limitációi miatt.

A Technológiai háttér fejezetben ismertetem a választott szoftveres technológiákat. Felsorolok egyéb, alternatív nyelveket és technológiákat is, amelyekkel ugyanezek a problémák megoldhatók.

Külön kitérek a véletlenszámgenerálás megvalósítására, mivel ez a hangyakolónia algoritmus egy központi eleme.

A Megvalósítás fejezetben fejtem ki, hogy a választott algoritmusok gyakorlati megvalósítását milyen módon végeztem. Mutatok példákat is kódrészletekre, amelyek valamilyen szempontból érdekesek. Megosztom a tapasztalataimat a nehézségekről, amelyekbe beleütköztem, vagy amik közvetve akadályozták a munkámat.

A kód bizonyos részegységei esetén több alternatív lehetőséget is bemutatok, amelyek máshogy közelítik meg ugyanazt a problémát.

Mivel az hangyakolónia és a hozzá hasonló heurisztikus algoritmusok működése több hangolható paraméterrel befolyásolható, így ezek jelentőségéről és a futásra gyakorolt hatásaikról is itt írok.

Az Eredmények fejezetben ismertetem a teszteléshez használt adatkészleteket, valamint a mérések módját és kitérek azokra a tényezőkre, amelyek a pontosságot ronthatták.

Prezentálom a referencia implementációhoz képes elért gyorsulást bemutató mérési eredményeket. Elemzem az eredményeket, valamint a paraméterek módosításának a futásidőre gyakorolt hatásait.

Becslést adok továbbá arra, hogy különböző paraméterek mellett mekkora adatkészletek kezelhetők.

Az Összefoglalás fejezetben röviden összegzem a dolgozatban véghez vitt munkát és értékelem a kapott eredményeket.

A Továbbfejlesztési lehetőségek fejezetben néhány példát mutatok arra, hogy a munkám hogyan folytatható tovább, akár az algoritmusok további gyorsításával, akár a hangyakolónia optimalizáció más problémakörökre való kiterjesztésével.

Felsorolom azokat a tervezési döntéseket, amelyek ahhoz kellettek, hogy minél sokoldalúbban felhasználható legyen az itt prezentált párhuzamos algoritmus.

2 Elméleti háttér

Ahhoz, hogy megfelelően működő, a feladatot megoldó kódot készíthessek, nagyon fontos a problémák megértése és az alkalmazott algoritmusok ismerete. Ebben fejezetben ismertetem a matematikai háttérüket és formalizálom az optimális megoldásokkal szemben támasztott igényeket.

2.1 Jármű útvonaltervezési probléma

A jármű útvonaltervezési probléma (vehicle routing problem, VRP) egy optimalizálási probléma, amely során járműveknek az útiköltségek minimalizálásával kell végig járniuk megadott állomásokat. Az alapproblémát gyakran kiegészítik egyéb megkötésekkel, mint például:

- járművek maximális száma
- egyes járművek szállítási kapacitása
- állomások lehetséges kiszolgálási időablaka
- egy jármű által megtehető maximális út

Amennyiben külön nincs más kikötve, a járművek ugyanabból a pontból indulnak, a kiindulási pontot leszámítva minden állomást legalább egyszer érinteniük kell és az útuk végén vissza kell térniük a közös kiindulópontba. A járművek minden állomáson pontosan egyszer rakhatnak le árut, a részleges kiszállítás nincs megengedve.

A problémát modellezhetjük egy irányított gráffal, amelyben a csúcsok az állomások, az élek pedig az egyes állomások közötti útvonalak. Az egyes élek súlya az út megtételének költsége (például távolság). Adott tehát

$$V = \{v_0, v_1, \dots, v_n\}$$

$$E = \{e_0, e_1, \dots, e_m\}$$

$$O \in V$$

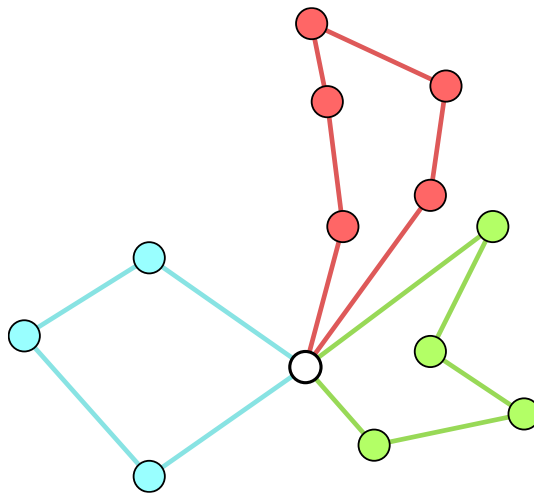
ahol V az állomások halmaza, n az állomások száma, E az állomások között elérhető utak halmaza, m az utak száma, O pedig a kiindulási állomás. Az élekre vezessük be ezt a jelölést: $e_{i,j}$ az az él, amely v_i -ből v_j -be tart, $w_{i,j}$ pedig ennek az útnak a költsége.

Adott továbbá a csúcshalmaz minden eleméhez egy d_i , az adott csúcs áruigénye, amit ki kell elégíteni.

A cél az, hogy útvonalak olyan R_i listáit állítsuk elő, ahol minden

$$R_i = (O, v_{j_0}, v_{j_1}, v_{j_2}, \dots, O)$$

egy jármű útvonalát írja le, amelyben az útvonalat alkotó élek $(e_{O,j_0}, e_{j_0,j_1}, e_{j_1,j_2}, \dots, e_{j_m,O})$. Az útvonal költsége az azt alkotó élekhez tartozó súlyok összege. A célunk azon R_0, R_1, \dots, R_k útvonalak megtalálása, amelyek összköltsége minimális és minden csúcst tartalmaznak, vagyis $\forall v_i \in V$ esetén $\exists R_j$, hogy $v_i \in R_j$.



2. ábra: Jármű útvonaltervezési probléma egy lehetséges megoldása

Az alapproblémára az optimális megoldás megtalálása NP-nehéz feladat, emiatt nagy csúcs- és élhalmaz esetén nem gazdaságos a kiszámítása. Annak érdekében, hogy a gyakorlatban használható algoritmust konstruáljak, valamilyen közelítő, heurisztikus megoldást érdemes használni a legjobb megoldást megtaláló algoritmusok helyett.

Egy lehetséges metaheurisztika a hangyakolónia algoritmus, amely amellet, hogy az optimálishoz nagyon közeli megoldások megtalálására képes, jellegéből fakadóan alkalmas a masszívan párhuzamos futtatásra. Ezt a tulajdonságát használom ki, amikor az Nvidia CUDA architektúrájára adaptálom az algoritmust.

2.1.1 Korlátozott kapacitású járművek

A VRP problémák során alkalmazható egy gyakori nehezítés az egyes járművek kapacitásának korlátozása. Ez azt jelenti, hogy az egyes járművek által érintett állomások

teljes áruigénye nem haladhatja meg a jármű kapacitását. A logisztikai problémáknál a kapacitás lehet például egy teherautó maximális terhelhetősége.

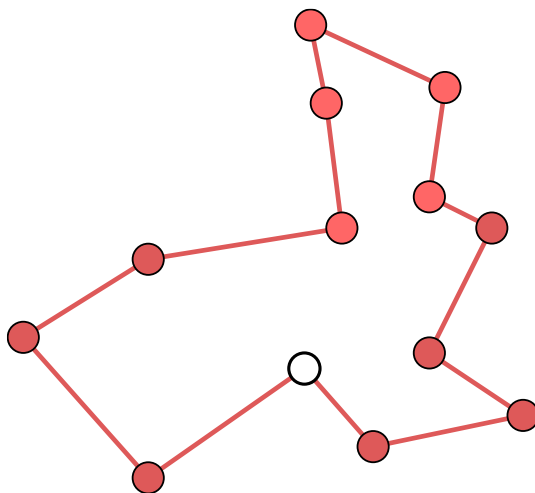
Innen kapta a VRP variáns a nevét: Capacitated Vehicle Routing Problem (CVRP). Én azt az egyszerűbb esetet fogom vizsgálni, ahol minden jármű azonos kapacitással rendelkezik. Jelöljük ezt a kapacitást C -vel! Ekkor az egyes útvonalakra teljesülnie kell, hogy

$$\sum_{v_j \in R_i} d_j < C$$

minden R_i -re.

2.2 Utazóügynök probléma

Az utazóügynök probléma tekinthető a jármű útvonaltervezési probléma azon speciális esetének, amikor a járművek maximális száma 1 és a kapacitás nincs korlátozva.



3. ábra: Utazóügynök probléma egy lehetséges megoldása

A feladat itt a legkisebb költségű Hamilton-kör meghatározása a gráfban, vagyis azon

$$R = \{O, v_{j_0}, v_{j_1}, v_{j_2}, \dots, O\}$$

bejárás meghatározása, amely V minden csúcsát tartalmazza és amelynek költsége minimális.

A VRP problémához hasonlóan a TSP is az NP-nehéz feladatok közé tartozik. A megoldás irányított gráf esetében nehezebb, mint a

2.3 Hangyakolónia algoritmus

A hangyakolónia algoritmus egy heurisztikus útkereső algoritmus. Működése a hangyák természetben megfigyelt élelemkeresési módszerén alapul. Az egyes hangyák véletlenszerű útvonalakon keresik az élelemhez vezető utat, majd a visszaúton a bolyba feromonnal jelölik meg az útvonalukat. Ez alapján a később induló hangyák a feromon erőssége alapján már informált döntéseket hozhatnak a bejárást illetően. A hátrahagyott feromon erőssége idővel gyengül, ezzel biztosítva, hogy a nem optimális útvonalak (az élelemhez vezet, de már van nála rövidebb) maguktól kikopjanak.

Az algoritmus futása során nyilván kell tehát tartani egy feromongráfot. A feromongráf és az eredeti gráf topológiája megegyezik, de az élsúlyok eltérőek. A feromongráf súlyait $\tau_{i,j}$ -vel jelöltem.

Egy csúcsból való továbblépéskor a hangyák a lehetséges kimenő élek közül a feromon és az élsúly alapján súlyozottan, véletlenszerűen választanak. Az egyes élek kiválasztásának valószínűsége

$$P_{i,j} = \frac{(\tau_{i,j})^\alpha (w_{i,j})^\beta}{\sum_{l \in N_i^k} (\tau_{i,l})^\alpha (w_{i,l})^\beta}$$

ahol N_i^k az i -edik csúcsból az algoritmus k lépésében elérhető szomszédos csúcsok halmaza. α és β a feromon és az élsúlyok szignifikanciáját szabályozza.

Az algoritmus minden iterációja során, miután a hangyák befejezték az egymástól független bejárást, a legkisebb költségű megtalált útvonal alapján frissítem a feromongráfot. Az egyes feromonértékekre két dolog van hatással: az idő múlásával minden feromon magától gyengül egy előre meghatározott konstans szorzóval, ez a párolgás (ρ). Emellett, ha az adott iterációban az él része volt valamely R útnak, akkor a feromonszintet növelem az út költségének reciprokával arányosan, tehát a kisebb költségű utakon erősebb feromont hagynak majd hátra a hangyák.

$$\tau_{i,j} \leftarrow (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}$$

$$\Delta\tau_{i,j} = \begin{cases} Q/W, & \text{ha } e_{i,j} \text{ része valamely } R \text{ útnak} \\ 0, & \text{egyébként} \end{cases}$$

ahol W az egy iterációja során megtalált legjobb R utak összköltsége, Q pedig a feromonfrissítést szabályozó konstans, amit mindig a bemenő adatoknak megfelelően kell meghatározni.

Az útkeresés közben mindig fel kell jegyezni a legjobb addigi megtalált utat. Az ACO algoritmus egyik előnye, hogy több, hasonlóan jó alternatív utat is képes megtalálni. Ez például térképes útvonaltervezésnél lehet hasznos.

A hangyakolónia alapú algoritmusok az útvonal optimalizálás mellett egyéb más problémák megoldására is alkalmazhatók, például ütemezési problémákhoz vagy képfeldolgozás során végzett éldetektáláshoz.

2.3.1 Alternatívák

A VRP és TSP problémák is megoldhatók egyéb módokon akár egzakt, akár közelítő megoldást keresünk.

Az egzakt megoldásra a legegyszerűbb, de naiv módszer az, ha minden egyes lehetséges bejárást legenerálunk és szimplán kiválasztjuk a legolcsóbbat, ami kielégíti a problémára vonatkozó egyéb megkötéseket. Ha az állomások n csúcsból álló teljes gráfot alkotnak, akkor TSP esetében $n!$ különböző bejárást kell megvizsgálni. Ez csak nagyon kis számú csúcs esetén számítható ki. A jármű útvonaltervezésnél az útvonal permutációk száma ennél is több, így ott fokozottan jelentkeznek a heurisztikus és közelítő algoritmusok előnyei.

Egy lehetséges, nem analitikus megoldás mindkét problémára a genetikus algoritmusok alkalmazása. Ehhez a lehetséges megoldásokat kölcsönösen egyértelmű módon le kell képezni genetikus kódokra, valamint az egyes kódokhoz meg kell tudnunk határozni egy úgynevezett fitnessz (fitness) értéket. Ez azt mutatja meg, hogy az adott kódhoz tartozó bejárást mennyire jó. Ebben az esetben érdemes az út hosszát választani fitnessznek. Genetikus kódot pedig például úgy képezhetünk a bejárásból, hogy felsoroljuk az érintett éleket. Ez az algoritmus úgy állít elő új megoldásokat, hogy a meglévők kódját kombinálja egymással és véletlen értékekkel, és mindig a legjobb fitnesszút kiválasztva ismétli a folyamatot, amíg valamilyen megállási feltétel nem teljesül (pl. elér bizonyos iterációt, fitnessz kellően jó). Mivel nem minden generált kód nem feltétlen teljesíthető bejárást ír le, így ezeket ki kell szűrni.

2.4 Adatreprezentáció

Az útvonaltervezési problémáknál az elérhető útvonalakat általában reprezentálhatjuk egy irányított gráffal. A gráf csúcsai azok a csomópontok, amelyek között útvonalat tudunk tervezni. A gráf élei és súlyai reprezentálják az egyik

csomópontból egy másikba való eljutásának lehetőségét, illetve költségét. A közlekedési útvonalak tervezésénél figyelembe kell venni, hogy ha „A” csomópont szomszédja „B” csomópontnak, az nem jelenti azt, hogy oda-vissza van köztük út (pl. egyirányú utca), ezért a gráf modellemet felkészítettem az irányított élek kezelésére is.

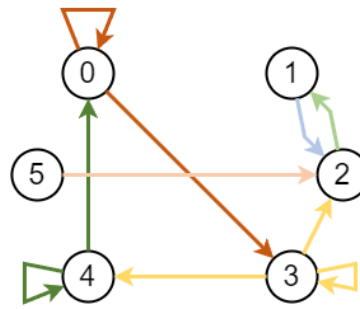
2.4.1 Szomszédsági mátrix

A gráfok modellezésének egy naiv módja a szomszédsági mátrix reprezentáció. Ennek előnye, hogy egyszerű, az egyes élekhez tartozó élek gyorsan lekérdezhetők, azonban a gyakorlati problémáknál helypazarló megoldás lehet. A közlekedési hálózatok esetén például a csomópontok száma gyakran nagyon nagy, viszont egy csomópont csak kevés másikkal kapcsolódik – ritka, hogy egy kereszteződésbe több mint 10 út torkollik. Ez azt eredményezi, hogy a mátrix ritka lesz, a legtöbb cellájában nem szerepel hasznos érték.

Mivel a videomemória mérete korlátozott, nem előnyös, ha üres értékekkel töltöm tele. Emiatt valamilyen hatékonyabb tárolási módot kellett találnom az adatok tárolására, ami kihasználja a gráf modell sajátosságait, a végső cél az, hogy a teljes probléma elférjen a grafikus hardver saját memóriájában, lehetővé téve, hogy több hangya útját szimulálhassam egyidőben akár nagyméretű adatkészlet esetén is.

2.4.2 Compressed Row Storage formátum

A Compressed Row Storage (CRS) [4] tárolási mód jelentősen csökkenti a gráf tárolásának tárigényét az élekhez tartozó súlyok indirekt elérése árán és a szomszédsági mátrixhoz hasonlóan alkalmas irányított gráfok tárolására. A módszer lényege, hogy két tömböt használok az élek, illetve az onnan elérhető csúcsok tárolására. A tömbök szerepét a gráf mátrix reprezentációjából kiindulva könnyű megérteni. Az első tömb érték párokat tárol. Minden érték pár egy él súlyából és az él cél csúcsának indexéből áll. A második tömb az egyes csúcsokhoz tartozó legelső él indexeit tartalmazza az első tömbben.



	0	1	2	3	4	5
0	7	0	0	2	0	0
1	0	0	2	0	0	0
2	0	4	0	0	0	0
3	0	0	5	5	3	0
4	5	0	0	0	6	0
5	0	0	2	0	0	0

index	0	1	2	3	4	5	6	7	8	9
oszlop_idx	0	3	2	1	2	3	4	0	4	2
súly	7	2	2	4	5	5	3	5	6	2

index	0	1	2	3	4	5
sor_ptr	0	2	3	4	7	9

4. ábra: Példa szomszédsági mátrixos és CRS tárolásra

Egy élhez tartozó súly a következő módon kérhető le, ha x indexű csúcsból keressük a súlyt az y indexű csúcsba. Az első tömböt érdemes két különálló, egyforma méretű tömbként kezelni.

```

keresett_súly = 0
első_él = sor_ptr[x]
utolsó_él = sor_ptr[x + 1] - 1
ciklus_él_idx = első_él-től utolsó_él-ig:
    ha oszlop_idx[él_idx] == y akkor
        keresett_súly = súly[él_idx]

```

Legyen ezek neve súly (súlyok), oszlop_idx (él hová tart). A második tömb neve legyen sor_ptr (él kiinduló csúcsa).

Ebben az esetben a 0 súllyal azt jelöltem, hogy x és y csúcsok között nincs él. Vegyük észre, hogy irányítatlan gráf esetén minden élt duplán tárolunk, fele ekkora tárigénnyel is tárolhatnánk ugyanekkora információt. Mivel azt szeretnénk, hogy az élek elérése a lehető leggyorsabb legyen, ezért a kétszeres memóriaigény elfogadható kompromisszumnak tűnik. Ha irányítatlan gráfot kellene tárolnom, akkor az élek egy részét Compressed Row Storage, másik részét Compressed Column Storage eljárással

lehet érdemes tárolni, hogy megmaradjon az elérés gyorsasága. A Compressed Column Storage hasonlóan működik a fent leírtakhoz, csak a sorok és oszlopok szerepe fel van cserélve.

Az sem jelent problémát, hogy egy élhez tartozó súly lekérdezése három memóriaműveletet igényel. A hangyakolónia algoritmus a gráf bejárása során egy csúcsból elérhető összes szomszédos csúcsok közül választ, ezért fontos, hogy a szomszédos csúcsok listáját könnyen megkapjuk. Szomszédsági mátrixos tárolás esetén ehhez a mátrix egész sorát végig kellene olvasnom, élek után kutatva, míg itt az oszlop_idx tömbből kiolvasható mely élek indulnak x csúcsból.

Csúcsok száma (N)	Élek száma (M)	Tárigény		Arány
		Ritka mátrix	CRS	
10	10	100	30	30,00%
100	100	10000	300	3,00%
1000	1000	1000000	3000	0,30%
10000	10000	100000000	30000	0,03%
10000	20000	100000000	50000	0,05%
10000	30000	100000000	70000	0,07%

5. ábra: Ritka mátrix és compressed row storage (CRS) alapú tárolás becsült tárigénye

Ennek a módszernek a további előnye, hogy a szimulációban használt további gráfok, például az élekhez tartozó feromonszintek és a feromonfrissítési gráf ugyanezzel a modellel tárolhatók, sőt az oszlop_idx, sor_ptr tömböket újra fel tudom használni, hiszen csak ott van szükség az előbb említett értékekre, ahol fut a csúcsok között él is.

Ugyanakkor azt is érdemes figyelembe venni, hogy sűrű mátrix esetén a CRS tárolás akár majdnem a duplájára is növelheti a tárigényt, így ebben az esetben előnyösebb a szomszédsági mátrixos tárolást alkalmazni. A CRS hatásfoka teljes gráf leírására lesz a legrosszabb, ilyenkor a tárigény közel kétszeres lesz.

Csúcsok száma (N)	Élek száma (M)	Tárigény		Arány
		Sűrű mátrix	CRS	
10	90	100	190	190,00%
100	9900	10000	19900	199,00%
1000	999000	1000000	1999000	199,90%
10000	99990000	100000000	199990000	199,99%

6. ábra: Ritka mátrix és (CRS) alapú tárolás tárigénye teljes gráf esetén

3 Technológiai háttér

A legfontosabb döntés, amelyet az algoritmus implementálása előtt meg kellett hoznom, a parallelizációs technológia kiválasztása volt.

3.1 Nyelv és párhuzamos technológia

A több, a piacon elterjedten használt lehetőség is létezik, amelyek közül a legszélesebb körben alkalmazott az Nvidia cég CUDA platformja és az OpenCL keretrendszer.

A kettő között a használat szempontjából a leginkább szembetűnő különbség az, hogy míg a CUDA csak egy interfészt (API-t) biztosít a grafikus hardver vezérlésére és GPU programok futtatására, addig az OpenCL önálló nyelv is. A CUDA hátránya, hogy csak az Nvidia bizonyos hardvereszközein érhető el. Mivel rendelkezem ilyen eszközzel, a választási szempontoknál elég volt a két technológia egyéb különbségeit figyelembe vennem.

A CUDA több programozási nyelvvel is használható, míg az OpenCL-hez el kell sajátítani egy szakterületspecifikus C-szerű nyelvet. Mivel már volt korábbi tapasztalatom a C++11 nyelvvel, ami hardverközeli módon teszi lehetővé a programozást, ezért az implementációkhoz ezt választottam. Ez a nyelv elterjedt és jól dokumentált, emellett az optimalizációk is hatékonyan kivitelezhetők, mivel a nyelv gépi kódra fordítását szabályozó szabvány jól ismert, és sok segédanyag elérhető az értelmezéséhez. Az Nvidia platformja más nyelvekkel is használható, többek között Python-nal is, azonban a korábbi tapasztalataim alapján a Python inkább alkalmas magas szintű, absztraktabb megvalósításokhoz, így a C++ mellett döntöttem.

Szintén a CUDA mellett szól, hogy sok dokumentáció és használati útmutató érhető el hozzá. Ezek egy részét maga a gyártó gondozza, míg másik része közösségi tartalom, fórumokon, blogokon, tanulmányokban érhető el.

Nemcsak a fejlesztés, de a hibakeresés is egyszerűbb, mivel az Nvidia több fejlesztői eszközt is biztosít, amikkel a helyes működés és a futási teljesítmény is nyomon követhető. A hibakeresés különösen kritikus a masszívan többszálú architektúrák esetén, mert egy hagyományos hibakereső nem képes értelmezni a GPU-n történő utasításvégrehajrást. Az OpenCL ebből a szempontból kevésbé kiforrott.

A CUDA további előnye, hogy több olyan szoftverkönyvtár is készült hozzá, amik jól használhatóak, megbízható működésűek és bonyolult feladatok elvégzését egyszerűsítik le. A teljesség igénye nélkül ilyen könyvtárak:

- cuBLAS: lineáris algebra
- cuRAND: véletlenszámgenerálás
- nvJPEG: JPEG kép dekódolás
- Thrust: gyakran használt algoritmusok (pl. rendezés, redukció, map művelet) GPU gyorsítással

Többek között ilyen az általam is használt cuRAND könyvtár, amellyel párhuzamos környezetben lehet igazoltan jó minőségű véletlenszámokat generálni. Ez egy fontos szempont volt számomra, hiszen a véletlenszám generálás korántsem triviális feladat.

3.2 Véletlenszám generálás

A számítógépes véletlenszám generálás egy olyan feladat, amelyre sok algoritmus támaszkodik. A teljesség igénye nélkül ilyenek bizonyos kriptográfiai algoritmusok, a Monte Carlo elven működő algoritmusok, zajgenerátorok. Ezen alkalmazásoknál különböző minőségű véletlenszám generátorokra van szükség. Valódi véletlenszámot csak valamilyen hardver-alapú entrópiaforrás segítségével állíthatunk elő (például radioaktív bomlás megfigyelésével). Természetesen ez a legtöbb problémához szükségtelenül költséges célhardvert igényel, a generált számsorozatok nem megismételhetők és a hordozhatóság is probléma lehet.



7. ábra: Véletlenszámgenerálás lávalámpák segítségével

3.2.1 Pszeudorandom generátorok

Elterjedt alternatíva a számítástudományban valamilyen pszeudorandom generátor (pseudorandom number generator, PRNG) alkalmazása, amely egy olyan algoritmust takar, ami képes egy kezdőértékből (seed) egymástól statisztikailag független véletlen értékek sorozatát (szekvenciát) előállítani. Egy pszeudorandom generátor adott seed-hez mindig ugyanazon szekvenciát állítja elő, a különböző seed-del generált szekvenciák nem feltétlenül függetlenek egymástól, azonban a szekvencián belül az értékek igen.

A pszeudorandom generátorok előnye tehát a hordozhatóságuk, a megismételhető viselkedés és a gyorsaság. Ez utóbbi tulajdonságuk különösen akkor lényeges, ha nagy mennyiségű véletlenszámra van szükség (például Monte Carlo módszereknél).

3.2.2 Véletlenszámgenerátorok vizsgálata

A PRNG algoritmusok megjelenésével természetes igény keletkezett az általuk generált számsorok „véletlenségének” vizsgálatára. Erre a feladatra többféle módszer és eszköz is készült, amellyel tetszőleges szekvenciák tesztelhetők. Ezek egy része közvetlenül a bitek/bájtok statisztikai tulajdonságai vizsgálja, más része a szekvenciát számok sorozataként értelmezi, és ezeket különböző matematikai konstrukcióban vagy fizikai szimulációkban felhasználva a kapott eredmény eltérését vizsgálja az ideális esetben elvárható értéktől. A statisztikai tesztek közé tartozik többek között az egyes értékek vagy rész-szekvenciák előfordulásának vizsgálata.

A szoftveres véletlenszám generátorok teszteléséhez készültek olyan szoftverkönyvtárak, amelyekkel bizonyos szekvenciákról nagy biztonsággal eldönthető, hogy véletlenszerűek-e. Az egyik ilyen szoftverkönyvtár a TestU01, amely különböző teszt sorozatokat tartalmaz, mint például a SmallCrush vagy a BigCrush. A TestU01 használati útmutatója [5] alapján egy jó PRNG-vel szemben nem elvárás, hogy teljesítse az összes tesztet minden seed értékre. A tesztek egy részére úgy hivatkozik, mint könnyű tesztekre, ezek a véletlenszámok gyakori felhasználási módjait szimulálják. Az útmutató szerint ezeken a teszteken elvárható egy jó generátortól, hogy mindet feltétlenül teljesítse.

Mivel az alkalmazott tesztek nem közvetlenül a generált bitek véletlenszerűségét tesztelik, hanem a nyers adatot számokként értelmezik, ebből kifolyólag a számábrázolási mód szignifikánsabbnak tekinthető bitjein bekövetkező hibákat nagyobb súllyal érzékelik, míg a kevésbé szignifikáns bitek hibájára kevésbé érzékenyek. Ha a PRNG által generált szekvencia nem a teszt által használt számábrázolás szerint kerül felhasználásra, akkor a tesztet fordított bájtsorrendben is le kell futtatni, hogy ezt a jelenséget kezelni lehessen.

3.2.3 CuRAND

GPU alkalmazásoknál az is fontos szempont a randomgenerátor kiválasztásánál, hogy a grafikus hardveren lehessen előállítani a szekvenciát, ne kelljen a GPU algoritmusnak a CPU-ra vagy külső véletlenszám forrásra várni, valamint, hogy párhuzamosan több szálon használható legyen.

Saját PRNG-t készíteni az esetek többségében nem éri meg, mivel már léteznek megbízható, tesztelt és jól dokumentált megoldások erre a feladatra, amik sok fejlesztési időt takarítanak meg a programozónak. CUDA platformon elterjedt a cuRAND könyvtár használata, amely többféle PRNG algoritmust is tartalmaz, így a legtöbb esetben elég ezek közül az adott problémának megfelelőt kiválasztani. Az algoritmusok között a praktikus különbség a garantáltan véletlen szekvenciák minimális hosszában, a párhuzamosság támogatásában, a generálás számítási igényében és abban van, hogy képesek-e a szekvenciából adott eltolással értéket visszaadni.

A CuRAND-ban implementált `Philox_4x32_10` generátor a Crush tesztek mindegyikét teljesíti [6], kellően hosszú szekvenciákat generál, illetve a cuRAND-on belül viszonylag gyorsnak mondható, ezért ezt használtam az implementáció során.

3.3 GPGPU programozás

A modern grafikus kártyák számítási kapacitása jócskán meghaladja a CPU-két, azonban a használatához az algoritmusokat fel kell készíteni a párhuzamos futtatásra. Az itt bemutatott működési elvek az Nvidia CUDA platformjára [7] vonatkoznak, más technológiák ettől eltérhetnek.

A GPU-k nagy számítási teljesítménye azon alapul, hogy a gyártók sok független számítási egységet helyeznek el egy chipen, amik párhuzamosan futtatják a kód egyes szálait. Az egyes szálak rendelkeznek saját memóriával és látják az egész GPU globális memóriáját is, így akár kommunikálni is képesek egymással vagy felhasználhatnak más szálak által kiszámolt adatokat. Azonban ahhoz, hogy ne léphessenek fel versenyhelyzetek, tudatos és szigorú konkurenciakezelésre van szükség.

A GPU-k két fontos hardveres limitációval rendelkeznek, amiket nem lehet kikerülni: a memória sávszélességgel (GB/s) és a maximális számítási teljesítménnyel (TFLOPS). A memória sávszélesség azt korlátozza, hogy mennyi adatot tudunk másodpercenként írni és olvasni. Akkor ütközhetünk bele, ha a program sok adatot mozgat, de kevés műveletet végez vele. Ha túlzottan memóriaintenzív az algoritmus, a számítási teljesítmény kihasználatlan marad, az aritmetikai egységek éheznek amíg a memóriaműveletekre várnak. Ezzel szemben, ha számítás-intenzív algoritmust futtatunk, az aritmetikai egységek folyamatosan dolgoznak és csak időnként történik memória elérés. A GPGPU (general purpose GPU, általános célú GPU) optimalizálás során az egyik célom az, hogy mindkét limitet a lehető leginkább megközelítsem, a rendelkezésre álló hardver erőforrásokat minél jobban kiaknázzam.

3.3.1 Optimalizálás memória sávszélességre

A memória sávszélesség kihasználására többféle gyakorlatot is alkalmaznom kellett. Többek között azt, hogy az adatot lehetőleg ott tároljam, ahol szükség lesz rá. A GPU-k többszintű memóriarendszerrel dolgoznak, a globális memóriájuk általában nagy, minden szál számára látható, azonban lassú. Az egyes szálak saját memóriája viszonylag kicsi, cserébe az itt végzett művelet időigénye is alacsony. Sok GPU rendelkezik a kettő között elhelyezkedő úgynevezett osztott (shared) memóriával, ez nem globális, hanem egy szál blokk (thread block) közösen gazdálkodik vele, elérési ideje és mérete is többnyire a szálankénti és a globális memória közé esik.

A memória kihasználás javítására érdemes tudatosan megtervezni az adatok memóriában való elrendezését is, hogy hatékonyan cache-elhető legyen az olvasás: azokat az adatokat, amik egyszerre fognak kelleni, egymás mellé érdemes elhelyezni a memóriában. Bizonyos esetekben érdemes lehet az adatokat padding-gel kiegészíteni, hogy az összetartozó adatok garantáltan egyszerre kerüljenek be a cache-be. Ez a tárigény növekedését okoz, így mérlegelni kell, hogy a kevesebb memóriaművelet okozta gyorsulás megéri-e a plusz előfeldolgozást és a nagyobb memóriaigényt.

3.3.2 Optimalizálás utasításvégrehajtás-kihasználásra

A számítási teljesítmény kihasználásáért a legtöbbet azzal tehetjük, ha gondoskodunk róla, hogy a GPU-nak mindig álljon rendelkezésre elegendő futtatható szál, ami épp nem memóriaműveletre vár. Mivel a GPU órajele jellemzően elmarad a CPU-tól, ezért ugyanaz a kód lassabb lesz. Ahhoz, hogy egy algoritmus futását fel lehessen gyorsítani, a feladatot szét kell bontanunk több párhuzamos szál által elvégezhető műveletekre. A grafikus hardver egyszerre megadott számú szálát tud futtatni, ha ennél többet indítunk, akkor a többlet szál várakozik, amíg az aktív szálak számítási műveleteket végeznek. A memóriaműveletek 10-200 órajelciklusig is tarthatnak, a memória típusától függően. Ezért, ha egy aktív szál memóriaműveletbe kezd, akkor a háttérbe kerül, míg a helyét egy nem aktív szál veszi át (a gyakorlatban nem egyesével cserélődnek a szálak, hanem csoportokban). A GPU ütemező így képes elrejteni a memóriaműveletek lassúságát, az aritmetikai egység soha nem “éhezik”, feltéve, ha mindig van elegendő alkalmas szál, amiből válogathat.

A GPU-ban több különálló végrehajtási egység van, ezeket streaming multiprocessor-nak (SM) nevezik. Egy SM egyszerre meghatározott számú szálát képes futtatni. Egy ilyen csoport neve warp. A jelenlegi CUDA verziók 32 szálból álló warp-okkal dolgoznak. A warp-on belül minden szál ugyanazt az elemi processzor utasítást hajtja végre (lock-step végrehajtás). Ennek az a következménye, hogy ha a warp-on belül a szálak végrehajtási útjában divergencia alakul ki például egy elágazás miatt, akkor az összes szálnak meg kell várnia minden végrehajtási ág kiszámítását. Ezt a fejlesztés során figyelembe kell venni, mert a felesleges várakozások rontják a grafikus hardver kihasználtságának hatásfokát.

Egy warp-on belül a szálak képesek speciális utasításokkal adatot cserélni egymással. Ez különösen hasznos a párhuzamos redukciók megvalósítása során, mivel

így az utolsó 32 szál adatait globális vagy osztott memória használata nélkül is el lehet végezni.

A GPU-knak vannak ezen kívül is limitációi (regiszterek száma, max thread/thread block), amelyekre most nem térünk ki részletesen, azonban a fejlesztés során ezeket is szem előtt kell tartani.

3.3.3 Memória limitáció

A grafikus hardver használható memóriája véges. A videomemória (VRAM) továbbá nem egy összefüggő memóriaterületként érhető el, hanem bizonyos részei különböző méretben állnak rendelkezésre a programok számára a korábban leírt módon. A fejlesztés során figyelembe kell venni, hogy az egyes szálak számára csak limitált memória érhető el.

A hagyományos RAM memóriával ellentétben a VRAM nagysága a grafikus hardvertől függ, emiatt nehezen bővíthető. Ha egy algoritmus tárigénye meghaladja az elérhető grafikus memóriát, akkor a problémát le kell egyszerűsíteni vagy részekre kell bontani, azonban a részekre bontás nem minden feladtnál lehetséges. Ha van is rá lehetőség, a RAM és a VRAM között az adatok mozgatása lassú, emellett a másolást szinkronizálni kell mindkét oldalon, ami többnyire a kernel futásának befejezését jelenti.

A jelenlegi legújabb CUDA verziókban is csak 512KB helyi memória áll rendelkezésre a szálak számára, azonban ez a teljes memória felhasználás függvényében kevesebb is lehet vagy korlátozhatja az egyszerre futtatott szálak számát.

3.3.4 Kernel indítás

A GPU által futtatható eljárásokat kernelnek nevezzük. A kerneleket a CUDA API segítségével a CPU kódból indíthatjuk el. A kernelhívás aszinkron művelet, nem blokkolja a főszál futását. Ahhoz, hogy a GPU műveletek eredményét hasznosítani tudjuk, meg kell várnunk a kernel befejeződését. Erre a platform több eszközt is biztosít.

A kernelindítások végrehajtási overhead-del járnak, ami több részből tevődik össze [8]. Az általam használt grafikus kártya esetén ez néhány mikroszekundum. Ezt az időt a CUDA több kernel egymás után történő indításával képes elfedni. Ennek jelenleg az a feltétele, hogy a kernelek egy stream-ben fussanak és az indításuk között az overhead-nél kevesebb idő teljen el.

A stream a CUDA terminológiában az egymás után szekvenciálisan, indítási sorrendben futtatott kerneleket takarja. Ha két kernel program különböző stream-en fut, akkor azokat egymással párhuzamosan képes végrehajtani a GPU különböző streaming multiprocessor-ok használatával.

Mivel a kernel indítások időtöbbletét a GPU elfedi, ezért érdemes inkább több, logikailag különálló kernelre bontani az algoritmust, mert így jobban kihasználhatók a hardver erőforrások és átláthatóbb a kód mintha egy nagy, univerzális kernellel dolgoznánk.

4 Megvalósítás

Az algoritmusoknak először az egyszálú implementációját készítettem el. Ez amellet, hogy lehetővé tette a helyességük ellenőrzését, kiindulási alapot szolgáltatott a CUDA implementációkhoz, valamint az összehasonlító mérésekhez.

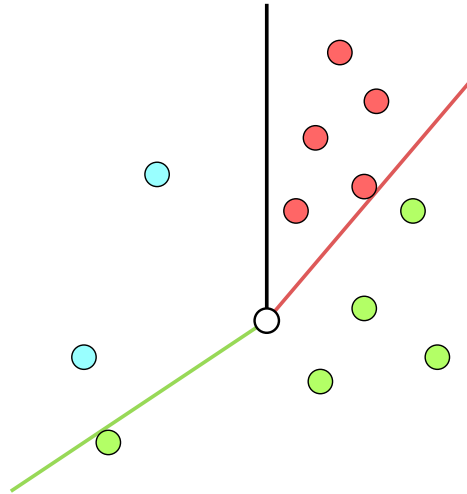
4.1 Hangyakolónia algoritmus

A TSP és a VRP algoritmus implementációja a párhuzamosítás szempontjából azonos, a VRP algoritmussal úgy lehet megoldani az utazóügynök problémát, hogy a jármű kapacitását az összes állomás áruigényét meghaladó értékre állítjuk. Ez elérhető egy kellően nagy szám alkalmazásával, vagy ha az áruigényeket nullának vesszük. Ebben az esetben egy jármű is elég, hogy kiszolgáljon minden állomást, a jármű így tulajdonképpen az utazó ügynök.

Ezen okból kifolyólag ebben a fejezetben kizárólag a jármű útvonaltervező algoritmusról fogok írni, a feltüntetett észrevételek érvényesek a TSP problémára is a fenti megkötésekkel.

Kezdetben az volt a szándékom, hogy az utazóügynök problémát oldom meg elsőként, majd arra az implementációra visszavezetem a VRP-t. Ennek az lett volna a feltétele, hogy az állomás gráfot felosszam valamilyen módon olyan összefüggő részgráfokra, amikben a csúcsok összesített áruigénye nem haladja meg a járművek kapacitását. Amennyiben ez sikerül, az egyes részgráfon már külön-külön alkalmazható a TSP-t megoldó algoritmus.

Ha az eredeti gráf teljes és a pontokról térbeli elhelyezkedéséről rendelkezésre áll információ, akkor a részgráfokra bontás megvalósítható poláris felosztással. Ennek lényege, hogy az O csúcsból induló félegyenesekkel úgy vágom a síkot részekre, hogy az egy síkrészbe eső csúcsok áruigénye ne haladja meg a járművek kapacitását.



8. ábra: Állomások csoportosítása polárkoordináták alapján

A gyakorlatban ezt úgy valósítottam meg, hogy indítottam egy félegyenest a kiindulási állomásból az y tengellyel párhuzamosan. Ezután a félegyenest addig forgattam O körül az óramutató járásával megegyezően, amíg az eredeti félegyenes és az elforgatott által kimetszett síkrészben levő állomások igénye túl nem lépte a kapacitást. Az addig érintett pontokból képeztem az első részgráf csúcsait. Az eljárást addig ismételttem, amíg minden pont bele nem került valamely részhalmazba.

Az első félegyenes irányának megváltoztatásával és a körüljárás megfordításával többféle felosztást is elő lehet állítani. A különböző felosztások eltérő legjobb utakat eredményezhetnek, ezeknek a kezelése további feladatot jelent.

Ennek a felosztásnak a használata azonban erős megkötésekkel is jár. Egyrészt csak olyan gráfokon működik, amelyekhez tartozik megfelelő térbeli adat. Ez nem szükségszerűen jelent síkbeli koordinátákat, az eljárás több dimenzióra is kiterjeszthető. Másrészt külön meg kell vizsgálni, hogy minden részgráf bejárható-e. Ha van olyan, amelyre ez nem teljesül, akkor az egész gráf sem bejárható a választott felosztás mellett. Minél ritkább egy gráf, annál kisebb valószínűséggel lehet felosztani így olyan módon, hogy minden részgráfban legyen Hamilton-kör.

Amennyiben ez sikerül is, nincs rá garancia, hogy az így kapott felosztáson futtatva az algoritmust, az képes lesz tetszőleges mértékben megközelíteni az optimális megoldást. Könnyedén előfordulhat, hogy a választott csoportosítás kizárja az legjobb bejárás által használt utakat.

Mivel azt szerettem volna elérni, hogy az algoritmus tetszőleges gráfon működjön, ezt a megközelítést elvettem. A VRP visszavezetése több kisebb TSP

feladatra önmagában is komplex feladat, ezért inkább úgy döntöttem, hogy a másik irányból közelítem meg az alapproblémát: egy olyan programot konstruáltam, ami a jármű útvonaltervezésre lett optimalizálva, azonban kis átalakítással képes megoldani az utazóügynök problémát is.

4.1.1 Egyszálú megvalósítás

Az ACO algoritmus implementálható egy szálon, ilyenkor a hangyák egymás után indulnak el és egy-egy csoport hangya sikeres útkereséséből álló iterációkat

Egy iteráció vázlata az alábbi:

- a feromon és a távolság gráf alapján kiszámítjuk, a csomópontok közti valószínűséget
- elindul egy csapat hangya
- a hangyák a csomópontok közül véletlenszerűen választva bejárják a gráfot
- a legjobb útvonalat megtaláló hangya feromont hagy maga után
- újakezdjük az iterációt

Az algoritmus pszeudokódja CPU megvalósítás esetén a következő:

```
ciklus 1-től n_iteráció-ig
    él valószínűség gráf kiszámítása

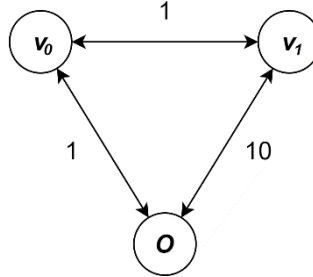
    ciklus 1-től n_hangya-ig
        útkeresés
        ha talált utat, akkor
            feljegyezzük a hangya útját
    ciklus vége

    feromongráf frissítése a legjobb talált útvonal alapján
ciklus vége
```

Fontos kiemelni, hogy abból kifolyólag, hogy a hangyák egy csúcsot csak egyszer járnak be, az algoritmus helyes működésének az a feltétele, hogy az egyes állomásokra a háromszög-egyenlőtlenség, vagyis a gráf bármely három olyan v_i , v_j , v_k pontjára, amelyek között fut él, teljesüljön, hogy

$$w_{i,j} < w_{i,k} + w_{j,k}$$

Amennyiben ez nem teljesül, az útkeresés még a legjobb esetben sem lesz képes megtalálni az optimális útvonalat. Ezt a következő egyszerű példa gráf alapján könnyű belátni:



9. ábra: Gráf, amelyben a csúcsokra nem teljesül a háromszög-egyenlőtlenség

Ha $d_0 = d_1 = 1$ és $C = 1$, akkor az algoritmusom által felfedezhető legjobb bejárás $R_0 = \{O, v_0, O\}, R_1 = \{O, v_1, O\}$. Ennek összköltsége $(w_{O,0} + w_{O,O}) + (w_{O,1} + w_{O,O}) = (1 + 1) + (10 + 10) = 22$.

Ezzel szemben, ha egy állomást többször is bejárhatna egy hangya, akkor a legjobb bejárás $R_0 = \{O, v_0, O\}, R_1 = \{O, v_0, v_1, v_0, O\}$ lenne, aminek az összköltsége $(w_{O,0} + w_{O,O}) + (w_{O,0} + w_{O,1} + w_{1,0} + w_{O,O}) = (1 + 1) + (1 + 1 + 1 + 1) = 6$.

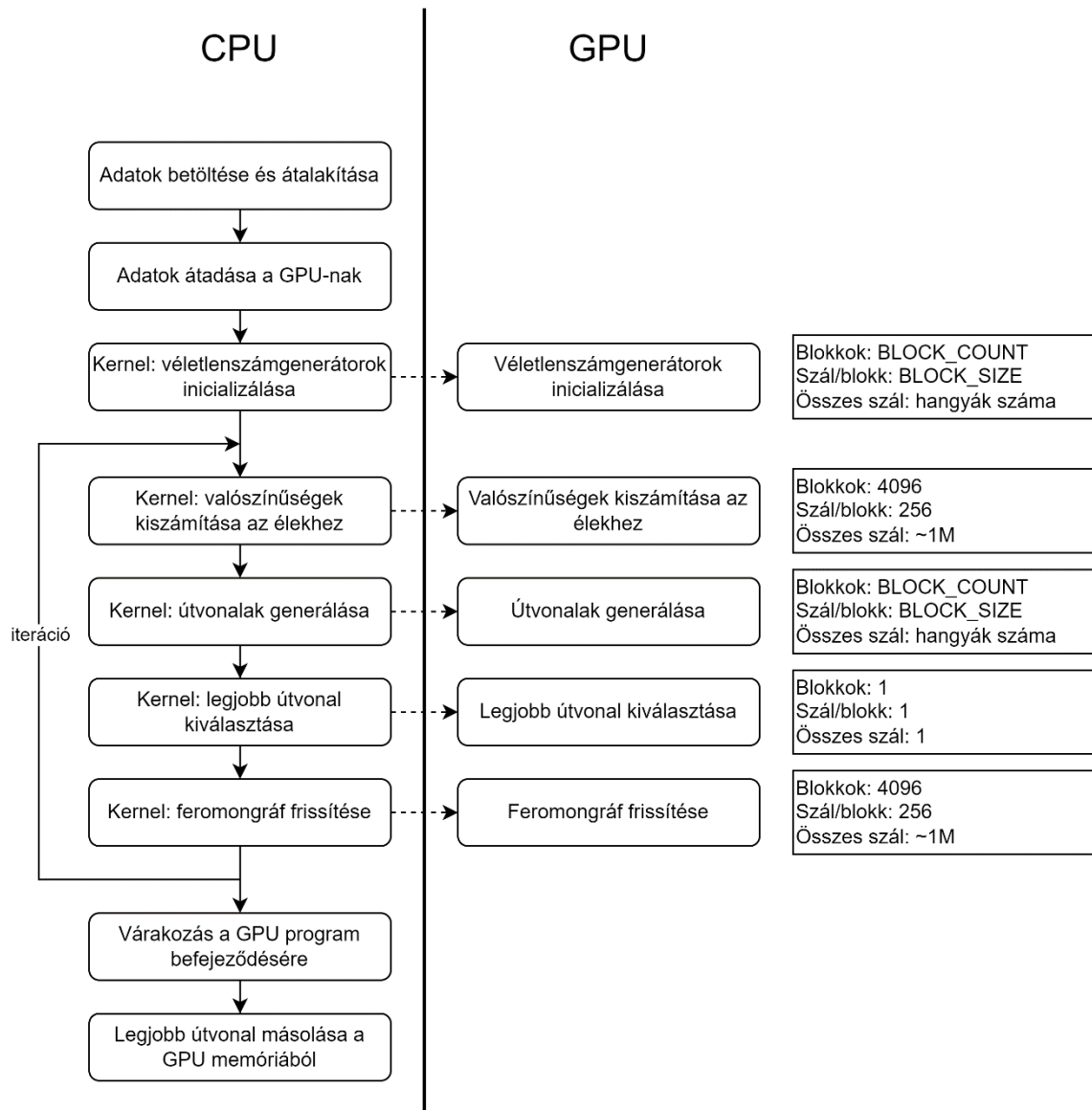
Mivel a következő él kiválasztása a többszálú megvalósítás esetén ugyanazokat az elveket követi, ezért a háromszög-egyenlőtlenség teljesülése ott is feltétel.

4.1.2 Többszálú megvalósítás

Az algoritmus jellegéből fakadóan masszívan párhuzamos futtatásra is alkalmas. Ez abból ered, hogy az egy csoportban induló hangyák útkeresése egymástól függetlenül is történhet, az iteráció kezdetén érvényes feromon és a valószínűségi adatokat leszámítva nincs szükség köztük információmegosztásra (egy iteráción belül). Ha a hangyákat a GPU program egy-egy szálához rendeljük, a működés sokszorosára gyorsítható ugyanakkora hangyaszám mellett.

Az első néhány GPU implementációhoz egy olyan kernelt konstruáltam, ami az egész feladat minden lépését elvégzi. Ezzel az volt a célom, hogy a kernelhívások okozta overhead-et kiküszöböljem. Ezt a megközelítést az első futásidő mérések után elvettem mivel láthatóan nagyon pazarlóan bánt a GPU erőforrásaival, valamint az algoritmus is szükségtelenül komplex lett tőle.

Arra a következtetésre jutottam, hogy a CPU program egyes logikai részeihez külön GPU kerneleket érdemes készíteni, mert eltérő párhuzamos algoritmusokat és más futtatási konfigurációt igényelnek. A program részegységekre bontása a Parallel CUDA Architecture for Solving de VRP with ACO publikáción alapul [9]. A párhuzamossá alakított algoritmus elvi felépítése a következő:



10. ábra: Párhuzamos implementáció elvi vázlata

A futás a CPU program elindításával kezdődik, ami betölti az adatokat és a megfelelő formátumba konvertálja őket, valamint inicializálja az algoritmus paramétereit. Az implementáció során az algoritmus CRS formátumban dolgozik a gráfokkal, így a bemenő adat struktúrájának függvényében szükség lehet átalakításra. Ezt az átalakítást a CPU-n végeztem, mivel ezen műveletek végrehajtását nem tekintem a

hangyakolónia algoritmus feladatának. Azért sem feltétlenül érdemes ezt a lépést a GPU-n végezni, mert kisebb méretű gráfok esetén ez a lépés elhanyagolható időt igényel az útkereséshez képes, míg a nagyobb gráfoknál az eredeti tárolási mód tárigénye meghaladná az elérhető VRAM méretét.

Ez után következik az adatok áttöltése a globális videomemóriába, továbbá a véletlenszámgenerátorok belső állapotának inicializálása. Mivel egy szál egy hangyának feleltethető meg, ezért fontos, hogy minden hangya tudjon saját, más hangyától nem függő véletlen értékek alapján utat választani.

A következő lépés a hangyakolónia algoritmus futtatása. A hangyák számát a BlockCount és BlockSize paraméterek változtatásával közvetve lehet megadni. Megegyező hangya mennyiségek többféle konfigurációval is elérhető. Ezek közül érdemes a leggyorsabb futást eredményezőt választani, ami adatkészletenként eltérhet.

BlockCount\BlockSize	32	64	128	256	512	1024
32	1024	2048	4096	8192	16384	32768
64	2048	4096	8192	16384	32768	65536
128	4096	8192	16384	32768	65536	131072
256	8192	16384	32768	65536	131072	262144
512	16384	32768	65536	131072	262144	524288
1024	32768	65536	131072	262144	524288	-
2048	65536	131072	262144	524288	-	-

11. ábra: Hangyák száma a paraméterek függvényében

Az algoritmus egyes részegységei külön kernelekben kerülnek meghívásra, mivel eltérő futási konfigurációra van szükségük. A kernelek szigorúan egymás után futhatnak csak, minden lépésnek szüksége van az azt megelőző kernel számításaira. Az iteráción belül tehát ez egy kritikus út.

4.1.3 Valószínűségek kiszámítása az élekhez

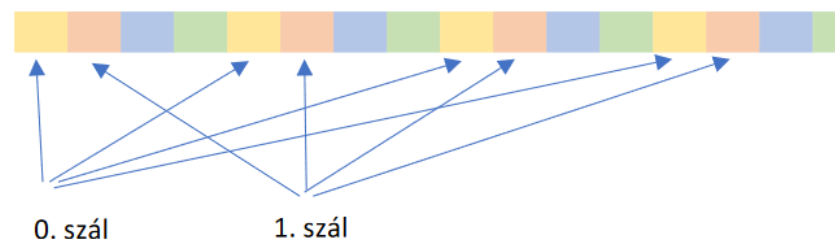
Minden élre meg lehet határozni az aktuális iterációban érvényes relatív valószínűségét. Érdemes ezt minden iteráció elején előre eltárolni, mivel az útkeresés során gyakran lesz szükség rá. Egy él relatív valószínűsége

$$p_{i,j} = (\tau_{i,j})^\alpha (w_{i,j})^\beta$$

Ez a képlet eltér a korábban bemutatott abszolút valószínűségtől. Ennek az az oka, hogy a relatív valószínűség nem tartalmazza azt az információt, hogy a hozzá tartozó él kezdőcsúcsából mely egyéb csúcsok érhetők el a bejárás jelenlegi lépésében.

A valószínűségek tömbjét a feromon és a költség tömbökön végrehajtott map művelettel kaphatjuk meg. A map művelet a bemeneti listák minden elemét (elempárját ebben az esetben) a kimeneti lista egy elemére képezi le. Az egyes leképezések egymástól függetlenül és tetszőleges sorrendben hajthatók végre, így nagyon hatékonyan párhuzamosíthatók.

A parallelizáció során a stratégiám az volt, hogy sok szálat indítok egyszerre és a GPU-ra bízom az ütemezésüket. Ha az élek száma meghaladja az általam indított szálak számát, akkor egyes szálaknak több él valószínűségét is ki kell számítaniuk, ezt a szálak számával való ismételt eltolásokkal oldottam meg. Ezt a módszert grid-stride ciklusnak [10] is nevezik.



12. ábra: Grid-stride 4 elem eltolással - minden szál több elem kiszámításáért is felelős

A memóriaműveletek során érdemes kihasználnunk a lokalitást. Mivel a globális memória olvasási és írási sebessége relatíve alacsony, ezért a GPU egyszerre több adatot is beolvas a gyorsítótárba. Ha a futtatott szálak számát ehhez igazítjuk, akkor az egy időben futó szálak a cache-ben tárolt adatokkal tudnak dolgozni. Ha az alkalmazott eltolás ettől eltérő, az nagyban rontja a memória sávszélesség kihasználtságát. [11]

A map leképezésben használt függvény meglehetősen sok processzorműveletet igényel, ezért ennek a kernelnek a futását leginkább a GPU magok számítási kapacitása korlátozza, míg az elérhető memória sávszélesség kihasználatlan marad.

Ha több szálat indítunk, mint ahány élt tartalmaz a gráf, akkor a szálak egy része nem végez hasznos számítást, csak az erőforrásokat foglalja feleslegesen. Ezt korlátozott erőforrásokkal rendelkező rendszer esetén figyelembe kell venni. A hangyakolónia

algoritmusnál ez nem okoz problémát, mert az iteráción belül a kernelek futásának átlapolására egyébként sincs lehetőség, így ez a kernel ilyen módon nem vesz el az algoritmus más kernelje elől erőforrást.

4.1.4 Útvonalak generálása

Minden hangya egymástól független módon bejárja a gráfot a kezdőcsúcsból kiindulva. Az érintett csúcsokat és az összköltséget folyamatosan számontartom.

A következő csúcs kiválasztásánál a lehetséges csúcsnak meg kell felelnie ezeknek a feltételeknek:

- vezet hozzá él a jelenlegi csúcsból ÉS
- még nem érintette a hangya az iterációban ÉS
- ha kiszolgáltatná a hangya az igényét, akkor sem haladnánk meg a jármű kapacitást

Azok közül a csúcsok közül, amelyek mindhárom feltételnek eleget tesznek, a hangya a valószínűségi gráf alapján a rulettkerés-algoritmus segítségével választ véletlenszerűen. Ha nincs több, a feltételeknek eleget tevő szomszédos csúcs, akkor vissza kell térnie a kezdőcsúcsba. A hangya útjának az előző visszaérkezés óta megtett szakasza egy jármű útjának felel meg. Ezzel kikényszerítem, hogy minden csúcsot pontosan egyszer érintsünk a gráfban a bejárás során. Ha minden csúcsot bejárt, a hangya végzett.

A kezdőcsúcsba visszatérés során meg kell határozni a visszatérés költségét. Mivel nem garantált, hogy minden csúcsból vezet él a kiindulóba, ezért az adatok előkészítése során egy segéd tömbben előre kiszámítottam minden egyes csúcsra a minimális visszatérési költséget. Ehhez érdemes a konkrét problémához legjobban illeszkedő algoritmust használni, például teljes gráf esetén szimplán az élsúlyt. Ha egy csúcsból a bejárás során visszatérünk, az nem azt jelenti tehát hogy a járműnek közvetlenül kell a kezdőcsúcsba visszamennie, hanem hogy a lehető legrövidebb úton kell visszatérnie. Ezen út meghatározása nem soroltam jelen algoritmus feladatai közé.

Azért építettem bele ezt a lépést a programba, mert ritka gráfok feldolgozása esetén gyakran előfordult, hogy egy jármű olyan zsákutcába jutott, ahonnan nem volt képes visszatérni egy lépésben a kezdőpontba. Ez azzal járt, hogy a hangyák nagy hányada nem tudta befejezni a bejárást. Azért, hogy ezt a jelenséget enyhítsem, a

hazatérés megkönnyítésére vezettem be a fenti módszert. Persze ilyenkor is előfordulhat, hogy a hangya olyan csúcsba téved, ahonnan nem tud visszatérni. Ezeket a kódban zsákutcának (deadEnd) neveztem. Ha egy hangya zsákutcába jut, akkor nem folytatja tovább a bejárást. Teljes gráfok feldolgozása esetén természetesen nem fordulhat elő ilyen, hiszen bármely csúcs bármely másikból közvetlenül elérhető.

Az egyes hangyák által bejárt útvonalakat egy-egy tömbben tárolom. A tömböket az algoritmus futtatása előtt lefoglalom a GPU globális memóriájában. A szükséges területet a konkrét gráf ismerete nélkül a legtöbb lépésből álló útvonal meghatározásával becsültem felül. Ha minden állomást külön járműnek kell kiszolgálnia, a hangya által végzett bejárás akkor fogja a legtöbb csúcsot tartalmazni, mivel minden egyes csúcs után vissza kell térnie O kezdőcsúcsba. Ekkor a meglátogatott csúcsok száma $2n + 1$. A konkrét probléma ismeretében ez a szám kisebb is lehet, de nagyobb soha.

Az útvonalkereső kernelnek mind a végrehajtási, mind a memória kihasználása meglehetősen alacsony.

A végrehajtási kihasználtságért elsősorban a bonyolult vezérlési struktúra és a sok elágazás felelős. A GPU szempontjából az az ideális, ha a warp-on belül minden szál mindig ugyanazokat az utasításokat hajtja végre, mint például a map leképezésnél. Itt ezzel szemben az egyes szálak erősen különböző végrehajtási utakat járnak be. Mivel a warp-on belüli szálak lock-step módban futnak, ezért egy szál még akkor is kénytelen a feltételes műveleteket végrehajtani, ha egy elágazás feltétele számára nem teljesül.

Továbbá az is rontja a hatásfokot, hogy az egyes szálak nem ugyanannyi idő alatt futnak le. A gyorsabb szálaknak várakozniuk kell amíg a leglassabb szálak nem végeztek.

A teljes memória sávszélesség a véletlen bejárásnak köszönhetően nem használható ki. Mivel a következő csúcs lépésről lépésre előre nem látható módon történik, ezért a cache-be gyakran kell új adatot mozgatni a globális memóriából, kevés a folytonosan olvasott adat. Ezt a jelenséget valamelyest csökkenteni lehetne, ha segítenénk az egy csúcshoz tartozó kimenő élek egyszerre való betöltését. Ezt úgy tehetnénk meg, ha a CRS konverzió során az egyes csúcsokhoz tartozó tömbelemek számát a memória cache blokkok méretéhez igazítanánk üres elemekkel való feltöltéssel (padding).

index	0	1	2	3	4	5	6	7	8	9
oszlop_idx	0	3	2	1	2	3	4	0	4	2
súly	7	2	2	4	5	5	3	5	6	2

index	0	1	2	3	4	5
sor_ptr	0	2	3	4	7	9

index	0	1	2	0	3	0	4	5	6	0	7	8	9	0
oszlop_idx	0	3	2	0	1	0	2	3	4	0	0	4	2	0
súly	7	2	2	0	4	0	5	5	3	0	5	6	2	0

index	0	1	2	3	4	5
sor_ptr	0	2	4	6	10	12

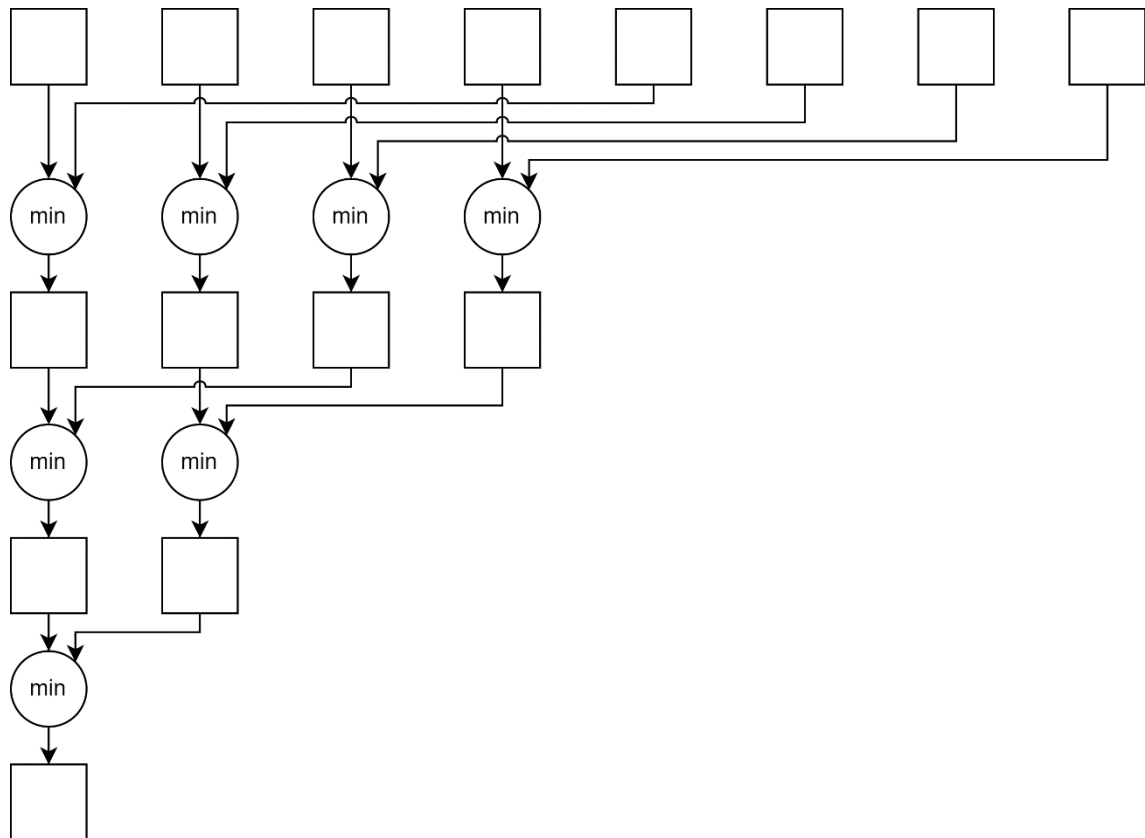
13. ábra: Eredeti CRS és paddinggel 2 változó szélesre kiegészített CRS tárolás

Ez a módosítás azonban amellet, hogy növeli az algoritmikus komplexitást mind az előfeldolgozásnál, mind a kernelben, csökkenti a hasznos memória méretét is. Minden padding kihasználatlan memóriaterületet jelent. Nagyméretű feldolgozandó gráfok esetén ez az overhead jelentős is lehet. Ezen megfontolásból kifolyólag ezt az optimalizációt nem építettem be az algoritmusba.

4.1.5 Legjobb útvonal kiválasztása

A feromonfrissítéshez ki kellett választanom a megtalált útvonalak közül a legjobbat. Erre egy egyszerű egyszálú minimumkereső algoritmust használtam. Egy ciklussal végig iteráltam az egyes hangyák által megtalált útvonalakon és a legjobbat helyi változóban tároltam. Ennek a lépésszáma a hangyák számával arányos.

Ennél a lépésnél a gyorsabb futás érdekében megfontoltam a párhuzamos redukció alkalmazását. Ennek lényege, hogy a szálakat párba állítjuk és minden párban meghatározzuk melyik a kisebb útvonalat tartalmazó szál. Ennek az adatait ezután átmásoljuk a másik szálba. Ezután elég az eredeti thread-ek felével tovább dolgoznunk, mert ezekben már megtalálható egy lokális minimum. A párosításokat addig ismételtetjük, amíg egyetlen szál marad. Ez fogja tartalmazni a globális minimumot.



14. ábra: Minimumkeresés párhuzamos redukció alkalmazásával 8 szál esetén

Könnyen belátható, hogy a redukció lépésszáma a szálak számának kettő alapú logaritmusával arányos amíg a redukálandó adatok száma meg nem haladja a szálak számát. Ezt túllépve azonban a grid-stride technikát alkalmazva kell előállítani annyi lokális minimumot, hogy az egyes szálakon párhuzamos redukciót végezhesünk.

Mivel a legjobb útvonal kiválasztása a teljes futásidőnek nagyon kis százalékát teszi ki már kis gráfok esetén is, ezért ezt az optimalizációt végül elvetettem.

Az ACO algoritmusnak létezik olyan változata is, amely nem csak az iterációnkénti legjobb utat veszi figyelembe a feromonfrissítésnél, hanem minden olyan hangya bejárt útvonalát, ami végül célba ért. Ezt a variánst nagy méretű, ritka mátrixok esetén érdemes alkalmazni, mert ott a hangyák sokkal nehezebben találnak alternatív útvonalakat és az általam készített implementáció sokkal több iterációt igényel. Ha a másik algoritmus változatot használtam volna, akkor a legjobb útvonalat kiválasztó kernelre nem lett volna szükség, a feromonértékek frissítése azonban valamivel komplexebb megvalósítást igényelt volna.

4.1.6 Feromonértékek frissítése

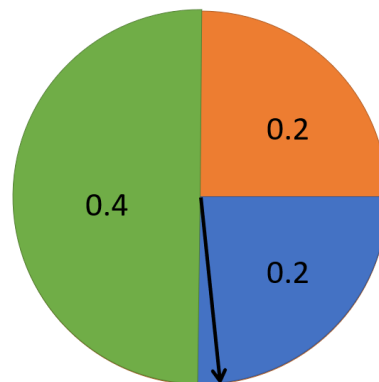
Ebben a lépésben a valószínűségeket kiszámító kernelhez hasonlóan egy map művelettel frissítenem kellett a feromonértékeket a korábban bemutatott képlet alapján.

A célom itt is, hogy minél több GPU szálát allokáljak és a grid-stride módszer felhasználásával mindegyik szál néhány él feromonértékét frissítse. Annak meghatározására, hogy az adott él része-e a legjobb útvonalnak, lineáris keresést használtam. Ha a vizsgált élt tartalmazta a legjobb út, akkor növeltem az út hosszának megfelelően, ellenkező esetben pedig csak a párolgást szimuláltam.

Mivel ez a kernel is a teljes futási időnek csak nagyon kis töredékét teszi ki, ezért további optimalizációkat nem alkalmaztam ennél a lépésnél.

4.2 Rulettkerék algoritmus

A hangyakolónia algoritmus a gráf bejárása során a csúcsokban döntést kell hozzon, hogy merre haladjon tovább az elérhető csúcsok közül. A választást az élek feromonértékeivel súlyozottan, véletlenszerű módon kellett, hogy végezzem. Ehhez a rulettkerék algoritmus használtam. Ennek a lényege, hogy egy képzeletbeli intervallumot (vagy rulettkereket) felosztunk a lehetőségeknek megfelelő rész-intervallumokra, amelyek aránya megegyezik a súlyok arányával. Ezután véletlenszerűen választunk egy értéket az intervallumból és megnézzük melyik szekcióba esik.



15. ábra: Véletlen súlyozott választás 3 lehetséges élből rulettkerék módszerrel

4.2.1 Rulettkerék algoritmus a gyakorlatban

A gyakorlati megvalósítás során természetesen szó sincs semmiféle kerékről. A gráf bejárása közben egy adott v_i csúcsban tartózkodva a CRS tárolásnak köszönhetően

rendelkezésemre áll az egyes élek relatív valószínűségeinek előre kiszámított tömbje, ahol az egyes csúcsok összes kimenő éle szomszédos elemek sorozatát alkotja. Ezen túlmenően ismerem az aktuális hangya által már bejárt csúcsok listáját ($V_{bejárt}$), valamint a jelenlegi jármű által eddig kielégített összes igényt (L) is. Mivel a kezdőcsúcsba való visszatérést külön kezelem, ezért nem veszem figyelembe a választás során az ide tartó éleket. Amikor következő állomást választok, a bejárt csúcsokat természetesen felesleges újból meglátogatni, valamint azokat is, amelyek áruigénye meghaladja a fennmaradó kapacitást, formálisan:

$$e_{i,j} \in E$$

$$V_j \neq V_S$$

$$V_j \notin V_{bejárt}$$

$$L + d_{i,j} \leq C$$

Jelöljük azon élek halmazát, amelyek eleget tesznek a fenti feltételeknek, E_{alk} -kal!

A fennmaradó élek relatív valószínűségeit tehát egy ciklussal összegzem, valamint feljegyzem az utolsó ilyen él indexét, mert erre egy ritkán előforduló szélsőséges eset kezelésénél szükség lesz. Ez az összeg

$$P = \sum_{e_{i,j} \in E_{alk}} p_{i,j}$$

Ezután generálok egy véletlenszámot, ami majd eldönt merre menjen tovább a hangya. Itt kézenfekvő lenne a $[0; 1]$ intervallumba eső számot előállítani és P -t, valamint az áruigény értékeket normalizálni, azonban ennél valamivel kevesebb aritmetikai utasítással jár, ha úgy generálok véletlen r értéket, hogy az a

$$r \in [0, P[$$

intervallumba essen.

Ebben az esetben egy ciklussal egyszerűen, lineáris kereséssel választom ki, hogy melyik az az $e_{i,j}$ él, amelyre teljesül, hogy

$$\sum_{e_{i,j} \in E_{alk}} p_{i,j} < r$$

úgy, hogy j maximális.

A lebegőpontos kerekítések miatt azonban figyelembe kell venni, hogy ha a lineáris keresés során elértem az utolsó alkalmas élt, akkor még abban az esetben is ezt az élt kell kiválasztani, ha a relatív valószínűségek összege nem éri el r értékét. Ennek elmulasztása hibás működéshez vezethet.

Amennyiben nem találok megfelelő csúcsot ahová tovább haladhatna a hangya, úgy vissza kell térnie a kiindulási csúcsba és ezzel az adott jármű útja véget ér. Ha maradt feldolgozatlan csúcs, akkor a hangya folytatja az útját, egyébként ez az iteráció végét jelenti a hangya számára.

A fent leírttal ellentétben a cuRAND működése miatt valójában nem a $[0, P[$ intervallumba eső véletlenszámot állítok elő, hanem a $]0, P]$ intervallumba, azonban ez annyira kis különbséget jelent a gyakorlatban, hogy a hatásától egyszerűen eltekintettem.

4.3 Legjobb útvonal meghatározása

Az útkereső algoritmus futása során a szimulált hangyák egyre jobb és jobb útvonalakat találnak, azonban ezt az információt ki kell juttatni a grafikus hardverből. A fentebb bemutatott, általam végül implementált algoritmus determinisztikus módon teszi ezt, azonban nem ez az egyetlen lehetőség a legjobb megtalált út meghatározására. Bemutatok egy eltérő módszert is, ami az ACO működésének egy sajátosságára épül.

4.3.1 Determinisztikus módszer

A legjobb útvonal meghatározható a futás közben a következő naiv determinisztikus megközelítéssel: futás közben nyilvántartom az addigi legjobb útvonalat és annak hosszát a GPU globális memóriájában. Ha találok rövidebb útvonalat az eddigi legjobbnál, akkor lecserélem a tárolt utat az újonnan felfedezettre és frissítem a legjobb költség változót is. Ilyen módon, a hangyák által valaha bejárt összes útvonal közül determinisztikus módon meg lehet határozni melyik volt a legjobb.

Ennek a módszernek azonban a párhuzamosítás szempontjából némi költsége van, mivel a szálak közül ki kell választanunk a legjobb útvonalat megtalálót. Mivel az általam választott implementáció egy iterációból csak a legjobb utat használja a feromonfrissítéshez, ezért ebben az esetben a teljes futás legjobb útvonalának meghatározása nem növeli számottevően a lépésszámot.

Ha ettől eltérően minden célba találó hangya útját figyelembe venném a feromongráf frissítésbe, akkor viszont a legrövidebb utat meghatározó kernel egyetlen célja az algoritmus végeredményének meghatározása lenne.

4.3.2 Probabilisztikus módszer

A végső, legjobb utat azonban máshogy is ki lehet következtetni. Ehhez azt használtam ki, hogy a hangyakolónia algoritmus futása során a feromongráf az idő nagy részében viszonylag kevés állapot egyikében van. Megfigyeltem, hogy amikor az ágensek megtalálnak egy már kellőképpen jó utat, akkor több iteráció is eltelik úgy, hogy nem találnak jobbat. Ennek következtében a feromon a megtalált legjobb útvonal mentén lesz maximális a gráf egyes élein, míg a többi élen a párolgás miatt hamar lecsökken a szintje.

Ez egészen addig így marad, amíg egy hangya véletlen rá nem bukkan egy még rövidebb útra. Ez az esemény kimozdítja a feromongráft az állandósult állapotából, mert a megtalált új út egyre több és több hangyát fog vonzani. Minél közelebb kerül a megtalált legjobb megoldás az optimálishoz, várhatóan annál több idő telik el két különböző, addig legjobbnak gondolt útvonal megtalálása között. Ennélfogva, ha kellően nagy iteráció után megállítom az algoritmust, jó eséllyel a feromongráf alapján rekonstruálni tudom belőle az utolsó utat.

Miután elegendő iterációt futtattam, a CPU programban bejárom a gráfot. Elindítok egy hangyát a kezdőpontból és minden csúcsból a maximális feromonkoncentrációjú élt mohó módon kiválasztva haladok tovább egészen addig, amíg el nem érek a cél csúcsba. Ha ezzel a módszerrel nem jutok el a kívánt csúcshoz, az három dolgot jelenthet:

- a problémának nincs megoldása, tetszőlegesen sok iteráció futtatása sem vezetne eredményre
- a végzett ismétlések száma túl kevés volt, a probléma megoldható, de még nem találtam meg az első közelítő megoldást sem
- egy új megoldás megtalálása után a feromonszintek még nem stabilizálódtak az új út mentén

Az első esetet sajnos nem tudom kivédeni, ahogy a heurisztikus algoritmusok általában, úgy az ACO algoritmus sem biztosít semmilyen garanciát arra nézve, hogy

tetszőlegesen sok lépés elvégzése után meg tudja állapítani, hogy nem létezik megoldás. Ennek eldöntésére más módot kell találni.

A második esetet emiatt nem lehet az algoritmus futásának megfigyelésével elkülöníteni az elsőtől. Ahogy azt sem, így ezt sem lehet kivédeni a determinisztikus módszer alkalmazásával sem, hiszen, ha a hangyák nem találtak egyetlen lehetséges megoldást sem, akkor a legjobbat sem lehet kiválasztani közülük.

A harmadik jelenség azonban a determinisztikus metódusnál nem fordulhat elő, hiszen a legjobb addigi utat egzakt módon határozom meg az összes megtalált alternatíva ismeretében. Szerencsére ez a probléma a probabilsztikus meghatározás során is könnyedén orvosolható. Csupán futtatnom kell még néhány további iterációt és a feromongráf egy újabb állandósult állapotba kerül, ahol a feromonok alapján a bejárás már meghatározható.

A fentebb leírt mohó bejárás szükség esetén tovább finomítható úgy, hogy ahelyett, hogy mindig a legnagyobb feromonértéket választanám, a legjobb n lehetséges élből választok a mélységi bejárás során. Ezzel kiküszöbölhető azon esetek egy része, amikor a hangyák egyik megtalált út használatáról éppen áttérnek egy másikra és a feromonszintek változása miatt a legjobb út nem olvasható ki egyértelműen.

A módosított feromonfrissítési logikát és a probabilsztikus módszert használva a legjobb utat kiválasztó kernel teljesen kihagyható a programból. Mivel ezzel az előzetes mérések alapján maximum néhány százalékos gyorsulást érhetnék el, ezért végül a valamivel kiszámíthatóbban működő determinisztikus megközelítést alkalmaztam az implementáció végleges formájában.

4.4 Szálanként több hangya szimulálása

Mikor még nem rendelkeztem előzetes becsléssel a sikeres futáshoz szükséges hangyák számát illetően, attól tartottam, hogy minden GPU szálon csak egy hangyát szimulálva az ACO algoritmus nem fog kellően jó közelítéseket adni.

Hogy ezt megoldjam, az útvonalkereső kernelt úgy készítettem el, hogy egymás után több hangya útját szimulálja és a következő kernelnek minden szálból csak a legjobb utat adtam át a következő kernelnek. Ennek köszönhetően a hangyák maximális száma a grafikus hardver limitációitól kevésbé függött.

Ennek ellenére ezt módosítást is elvetettem, mivel a méréseim során rájöttem, hogy az általam vizsgált problémák esetén a videokártyámon indítható szálak száma sokkal több mint amennyi hangyára ténylegesen szükség van és szálanként egy hangya elegendő.

Mivel ez a módosítás jelentős számítási és memória többletet igényelt a szálanként vett részeredmények kiszámításához és tárolásához, ezért az algoritmus ezen változatánál sokkal lassabb futást tapasztaltam, ezzel szemben a kapott közelítés pontossága elhanyagolhatóan javult csak.

4.5 Algoritmus paraméterek

A hangyakolónia elvű útvonalkereső algoritmus – ahogyan jellemzően a heurisztikus algoritmusok – működése több paramétertől függ. Ezen paraméterek egy része a használt technológiából adódik, másik része olyan, amit érdemes finomhangolni az ideális futás eléréséhez.

4.5.1 Technikai paraméterek

4.5.1.1 Epsilon

A lebegőpontos aritmetika során két változó egyenlőségét, illetve egy változó nullítását a kerekítések miatt nem lehet a szimpla egyenlőség operátorral végezni. A gyakorlatban ez úgy oldható meg, hogy definiálunk egy konstanst, ami a legkisebb kezelt számegységnek feleltethető meg. Ha két lebegőpontos érték között ennél kisebb a különbség, akkor a két szám ekvivalensnek tekinthető. Ebből következik, hogy ha egy szám abszolút értékben kisebb, mint epsilon, akkor a szám gyakorlatilag 0. A konstans értéke az adott probléma által igényelt pontosságtól és az alkalmazott lebegőpontos típustól függ.

4.5.1.2 Warp size

A használt CUDA hardvertől függ. Jelenleg minden eszköz 32 szálát foglal egy warpba, azonban ez a jövőben változhat. Általában érdemes ezt fordítási idejű konstansként definiálni, mert bizonyos algoritmusok optimalizálása során már a lefordított kód is eltérhet annak függvényében, hogy mekkora a warp size. Az ezen dolgozathoz készült kód nem használja ki, hogy ez a paraméter fordítási időben elérhető, ezért lehetne használni helyette egy dinamikus lekérdezést is a CUDA API-n keresztül.

4.5.2 Hangolható paraméterek

4.5.2.1 Alpha, beta

A hangyák a bejárás során a feromonértékek és az élsúlyok alapján határozzák meg melyik élen mekkora valószínűséggel haladhatnak tovább.

Szemléletesen az alpha és a beta a feromonszintek és az élsúlyok szignifikanciájának feleltethető meg a választás során. Magas alpha érték esetén a hangyák jobban ragaszkodnak a már megtalált utakhoz. Magas beta esetén pedig előszeretettel választják majd a rövidebb éleket, mohóbb viselkedést eredményezve ezzel.

4.5.2.2 Rho

A hangyák után hátramaradt feromcsíkok párolgását szimulálom vele. Egy adott élen minden iteráció végén a feromonértéket a következő módon frissítem:

$$\tau_{i,j} \leftarrow (1 - \rho)\tau_{i,j} + \Delta\tau_{i,j}$$

Ennek köszönhetően a régi, már nem használt utakhoz nem fognak ragaszkodni a hangyák, hanem képesek lesznek rövidebbeket találni.

Ha túl magas, akkor a hangyák a jó utakat is hajlamosak lesznek elfelejteni, ellenben, ha túl alacsony, akkor a megtalált rövidebb útvonalak több iteráció alatt cserélik csak le a korábbiakat.

4.5.2.3 Iterációk száma

Hány ismétlés fusson az útkereső algoritmusból. Több iteráció jobb útvonalat eredményezhet, de az időigény is ezzel egyenesen arányosan növekszik.

A programot könnyű átalakítani úgy, hogy ne előre meghatározott iterációt futtasson, hanem a hangyakolónia algoritmus addig fusson, amíg a megtalált legjobb útvonal hossza egy bizonyos érték alá nem esik. Ez akkor lehet hasznos, ha rendelkezünk előzetes becsléssel az optimális út hosszáról és ezt szeretnénk adott mértékben megközelíteni. Ilyenkor is érdemes persze felső korlátot szabni az elvégzett iterációk számának vagy a futási időnek, mivel nem garantált, hogy az ACO valaha is el fogja érni a célként megadott összköltséget, ami végtelen futáshoz fog vezetni.

4.5.2.4 Q

A feromonfrissítés kiszámításához használt konstans. Egy megtalált út mentén az egyes éleken a feromonértéket az út hosszával fordítottan arányosan növelem. Különböző problémák esetén a gráf élein a költségek eltérhetnek, ezt lehet kompenzálni Q-val. Lehetséges lenne már az előfeldolgozás során az élsúlyokat Q-val korrigálni.

4.5.2.5 BlockCount, BlockSize

Azt lehet beállítani ezekkel a paraméterekkel, hogy hány CUDA szál blokk fusson párhuzamosan, illetve egy blokkban hány szál legyen. A kettő szorzata adja meg az összesen futtatott szálak számát. Mivel az algoritmust úgy implementáltam, hogy minden szál pontosan egy hangya útját szimulálja, ezért így lehet közvetett módon a hangyák számát is megadni.

Ezen paramétereket a hangyák kívánt számának és a használt hardvernek megfelelően kell beállítani, hatásukat a futásidőre a méréseknél külön elemzem. Erősen ajánlott a BlockSize paramétert a hardver warp méretének egész számú többszörösére állítani, mivel a GPU ekkora számú szállal dolgozik egyszerre, így akkor is kiegészíti a warp szálainak többszörösére a szálak számát, ha nem használjuk ki. Ilyenkor a maradék szál nem végez hasznos számítást, inaktív állapotban van.

5 Eredmények

A GPU-ra adaptált algoritmus futását összevethetjük az eredeti, egyszálú implementációéval. Érdekes továbbá megvizsgálni az algoritmus hangolható, futásidő szempontjából releváns paramétereinek hatását a futásidő változására.

A szimulációk a következő paraméterekkel rendelkező számítógépen futottak:

- CPU: Intel i5 10400 @ 2.9GHz
- GPU: Nvidia GeForce RTX 2070
- RAM: 16GB DDR4
- Operációs rendszer: Windows 11 22H2
- Fordító: nvcc V11.6.112

A mellékelt forráskódokban minden fájlban két különböző helyen, a „Tunable parameters” résznél található a hangolható paraméterek. Ezekkel lehet befolyásolni a program működését. Az adatkészlet elérési útját be kell állítani fordítás előtt a futtatást végző személy fájlrendszerének megfelelően. A fordítás az nvcc fordítóval végezhető. Ez a CPU és GPU kódokat is képes futtatható programmá alakítani. A fordítást ezzel a paranccsal végeztem:

```
nvcc vrp_gpu.cu -o vrp_gpu.exe
```

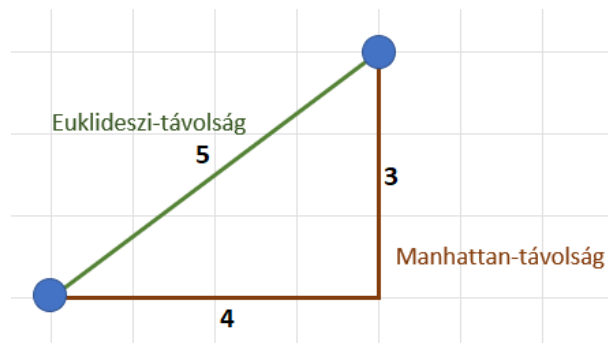
5.1 Adatkészlet

A mérésekhez az Éric D. Taillard [12] által közzétett adatkészletet használtam. Ebben a gyűjteményben 13 különböző fájl található. Minden adatkészlethez tartozik egy állomás halmaz koordinátákkal és áruigényekkel, amelyeket ki kell szolgálni. Ismert továbbá a járművek kapacitása és a kiinduló állomás.

Ezek alapján a VRP optimalizáció elvégezhető. Az eredmények kiértékeléséhez a szerző minden problémához csatolt egy ismert optimális megoldást is, ami alapján a heurisztikus algoritmusok pontossága mérhető. Ezt csak az algoritmus működésének ellenőrzéséhez használtam, a méréseknél nem vettem figyelembe.

A Taillard adatkészlet állomásai teljes gráfot alkotnak, bármely pontból eljuthatunk akármelyik másikba. Két pont távolságának az Euklideszi-távolságukat

tekintjük. Ezt azért fontos kiemelni, mert egyes adatkészletek a Manhattan-távolságot használják.



16. ábra: Euklideszi és Manhattan-távolság két pont között

Mivel az egyes állomások koordinátái egy sík pontjainak feleltethetők meg, ezért teljesül rájuk a háromszög egyenlőtlenség. Ez azért lényeges, mert az implementált ACO algoritmus kihasználja ezt a tulajdonságot, amikor a már meglátogatott csúcsokba nem tér vissza többször. Ha nem teljesül az egyenlőtlenség, akkor a következő csúcs kiválasztásánál nem zárhatnám ki a már bejártakat, mivel lehet, hogy rajtuk keresztül vezet a legrövidebb út egy még bejáratlan csúcsba.

Az adatkészlet többféle méretű adathalmazt tartalmaz, legtöbb méretből 4 különböző is rendelkezésre áll. Mivel ez összességében nagyon sok mérést eredményezett volna, ezért minden méretből egyet választottam ki. Ahol több lehetőség is rendelkezésre állt, ott az 'a' variánst használtam.

5.2 A mérések módja

A futásidők méréséhez a C++ `std::chrono::high_resolution_clock` osztályát használtam. Ennek a segítségével ezredmásodperces pontossággal mérhetjük az eltelt időt két tetszőleges időpont között. Mivel a méréseket Windows operációs rendszeren végeztem, ami nem valós idejű rendszer, ebből kifolyólag túl kis idők (nanosecundum) mérésére nem alkalmas, ezért olyan paramétereket választottam, ami legalább 100ms hosszú futást eredményez, így az időmérés pontatlansága és saját overhead-je elhanyagolható.

Mindkét implementáció esetében csak a tényleges algoritmus által igényelt időt mértem, az előfeldolgozást és az eredmény kiírását nem.

Nehézséget jelentett, hogy a CUDA kernelhívások aszinkron módon történnek, tehát a főszál alapértelmezetten tovább fut amíg a GPU számításokat végez. Külön paranccsal is ki lehet kényszeríteni, hogy a CPU várja meg az elindított kernelek befejeződését, de bizonyos utasítások, mint például a `cudaMemcpy(...)` ilyen szinkronizációt okoznak. A párhuzamos implementációnál a mérésbe beleszámoltam az adatok VRAM-ba másolását és az eredmény visszamásolását is.

Tudatosan döntöttem úgy, hogy a CUDA program futásába ezek is számítsanak bele, mivel arra voltam kíváncsi, hogy egy alapvetően CPU-n futó nagyobb program részeként hogyan teljesít a párhuzamosított algoritmus. Amennyiben csak a kernel programok futásidejét vizsgáltam volna, az Nvidia API-ja ehhez is biztosít eszközöket. A GPU kód futása során létre lehet hozni eseményeket (event) és az egyes események között eltelt időt ki lehet számítani. Ilyen módon nincs szükség a kernelt szinkronizálni még a mérés pillanatában sem, az események létrehozása is aszinkron történik.

Mivel a méréshez használt számítógépen más szoftverek is futottak, a háttérfolyamatok és egyéb futó programok befolyásolhatták az eredményeket. A mérésnél használt videokártya biztosította a monitor kimenetet is, ezért a tapasztalt teljesítmény várhatóan elmarad a maximálistól mind a kernelindítások időigénye, mind a kernelek futási ideje terén. Az Nvidia újabb modelljei képesek videokimenet nélküli módban, kizárólag CUDA eszközként működni így a Windows grafikus illesztő overhead-je elkerülhető. Ezt a faktort nem akartam minden áron semlegesíteni, mivel a monitor kimenetre szükségem volt a számítógép használatához.

A változó erőforráshasználat hatásának csökkentése érdekében minden beállítással 5 alkalommal futtattam minden algoritmust és az itt, a táblázatokban és a grafikonokon feltüntetett értékek ezeknek az átlagai.

Fontos megjegyezni, hogy a méréseknél kizárólag a futási időt vettem figyelembe, azonban vannak más olyan metrikák is, amelyekre egy gyakorlati alkalmazás során érdemes optimalizálni a kódot, például a VRAM sávszélesség és a GPU számítási teljesítményének a kihasználtsága, vagy a GPU kihasználtság (occupancy), ami azt mutatja meg hogy az elérhető warp-ok mekkora arányban voltak aktívak a program futása során. Egy igazán hatékony algoritmusnak nem csak gyorsan kell futnia, de a lehető legjobban kell gazdálkodnia az elérhető erőforrásokkal is.

5.3 Kiértékelés

Az elvégzett mérések eredményeiből több következtetést is levontam az algoritmus működésével kapcsolatban. Ezek egy része az ACO működéséből, míg másik része a feldolgozott konkrét problémákból adódik.

5.3.1 CPU és GPU összehasonlítása

A párhuzamosított algoritmus jóságának az eredeti, egyszálú megvalósításhoz mért gyorsulást tekintettem. Ez a mérőszám hasonló eredményeket mutatott a TSP és a VRP problémák esetén. A gyorsulás a két esetben nagyságrendileg megegyezett, az eltéréseket nagyrészt a véletlenség alapú útvonalválasztás okozta. A véletlenszámgenerátorokat más kezdőértékkel inicializálva a gyorsulás mértéke megközelítőleg azonos volt.

adatkészlet	állomások	hangyák	iterációk	CPU idő (ms)	GPU idő (ms)	gyorsulás
tai75a	75	1024	10	2656	302	8,79
tai100a	100	1024	10	5600	526	10,65
tai150a	150	1024	10	18421	1731	10,64
tai385	385	1024	10	282462	33732	8,37
tai75a	75	32768	10	82494	1848	44,64
tai100a	100	32768	10	185980	5114	36,37
tai150a	150	32768	10	617348	20807	29,67
tai385	385	32768	10	8958598	415325	21,57

17. ábra: Az algoritmus futásideje kis és nagy számú hangya használatával TSP problémánál

Az utazóügynök problémához a mérések szintén a Taillard adatkészlet használatával készültek. Itt nem állt ugyan rendelkezésre az optimális útvonal hossza, azonban az algoritmusnak elsősorban nem a becslési pontosságát akartam megtudni, hanem az elért gyorsulást. Ahhoz, hogy az utazóügynök problémára adjon megoldást, a programot úgy módosítottam, hogy az előfeldolgozás során minden állomás áruigényét nullára állítottam. Ennek következtében a hangya csak egyszer tért vissza a kiindulási csúcsba, vagyis a kapott egyetlen jármű útja az utazó ügynök útja.

adatkészlet	állomások	hangyák	iterációk	CPU idő (ms)	GPU idő (ms)	gyorsulás
tai75a	75	1024	10	2764	336	8,23
tai100a	100	1024	10	5946	584	10,18
tai150a	150	1024	10	18751	1876	10,00
tai385	385	1024	10	279461	36379	7,68
tai75a	75	32768	10	87714	2051	42,77
tai100a	100	32768	10	183155	4956	36,96
tai150a	150	32768	10	564490	17695	31,90
tai385	385	32768	10	8796533	351236	25,04

18. ábra: Az algoritmus futásideje kis és nagy számú hangya használatával VRP problémánál

Az eredmények értelmezésénél figyelembe kell venni, hogy a több maggal rendelkező CPU-kon is megvalósítható bizonyos fokú parallelizáció, és ez is valamekkora teljesítmény növekedéssel jár. Ehhez az értékhez képest a GPU használatával elérhető gyorsulás kisebb. Ugyanakkor még a legmodernebb processzorokban elérhető szálak száma is messze elmarad egy videokártyán futtatható szálak számától, így a CPU párhuzamosításnak komolyabb limitációi vannak a hangyák számát illetően.

Ahogy azt említettem, a Taillard adatkészletben bizonyos méretű gráfokból több különböző is rendelkezésre állt. Ezek közül mindig az 'a' variánst használtam, mivel az előzetes méréseim azt mutatták, hogy a gyorsulás szempontjából kicsi a jelentősége a konkrét adatoknak, inkább az állomáshalmaz mérete a meghatározó. Az azonos méretű gráfok 'a', 'b', 'c' és 'd' változatai az algoritmusok futásidejében nagyságrendileg 10%-os eltérést okoztak legfeljebb, azonban ez az eltérés egy konkrét gráfnál a CPU és a GPU futásidőt hasonló mértékben érintette. Ebből arra következtettem, hogy a gyorsulás a konkrét állomásoktól kismértékben függ csak.

5.3.2 BlockCount és BlockSize

Ugyanakkora számú hangyát többféle futási konfiguráció használatával is elérhető. Mivel a cél a minél rövidebb futás, azonos hangyaszám mellett érdemes azt a paraméter párosítást kiválasztani, ami a legkisebb futásidővel jár.

Ahhoz, hogy ezt megmérjem, többféle paraméterrel is lefordítottam a CUDA programot és megmértem az egyes változatok futási idejét. Ebből már ki tudtam számolni, hogy egy hangya futtatásának mennyi az átlagos költsége úgy, hogy a kapott időt elosztottam a hangyák számával.

BlockCount\BlockSize	32	64	128	256	512	1024
32	2,65820	1,38184	0,85327	0,65259	0,56006	0,62158
64	1,38867	0,85254	0,63611	0,53906	0,60477	0,62082
128	0,87305	0,63977	0,53949	0,60797	0,61275	0,62875
256	0,62732	0,54364	0,61188	0,60793	0,62319	0,64229
512	0,54102	0,60410	0,60593	0,61991	0,63761	0,61979
1024	0,53543	0,60574	0,61958	0,63463	-	-
2048	0,54611	-	-	-	-	-

19. ábra: Futási konfiguráció hatása a hangyánként vett időkölségre (ms/hangya)

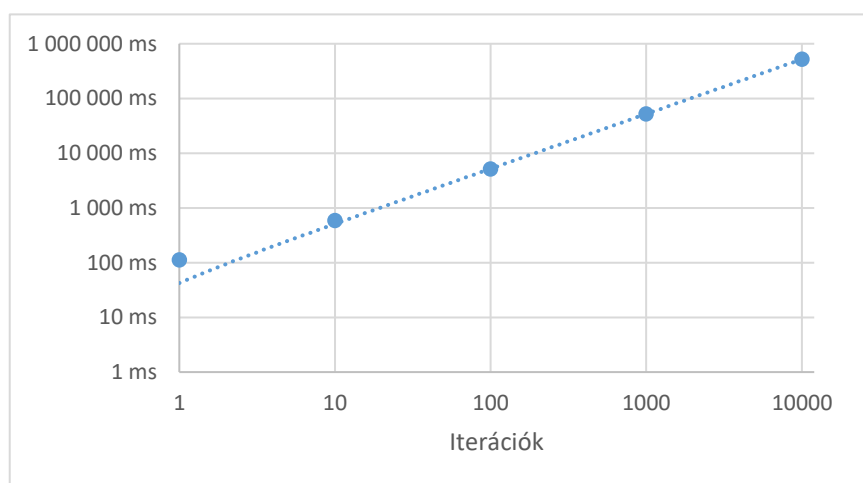
Látható, hogy ez az algoritmus ezen az adatkészleten hatékonyabb, ha több, kisebb méretű blokkot alokálunk a futtatáshoz. Ennek oka igen összetett, több dolog is befolyásolja, úgymint:

- véletlenszámgenerátorok kezdőértéke
- warp ütemező hardver működése
- adatkészlet

Ezen paraméterek összetett egymásra hatásai miatt nehéz általános tanácsot adni, ami minden esetben működik. Érdemes ezeket a paramétereket mindig a megoldandó problémához és a használt hardverhez igazítani.

5.3.3 Iterációk száma

Az ACO algoritmus pontosságát úgy is lehet javítani, ha több iterációnyi hangyát futtatunk. Ezzel nem növeljük a program memóriaigényét, azonban a végrehajtáshoz szükséges időt igen.



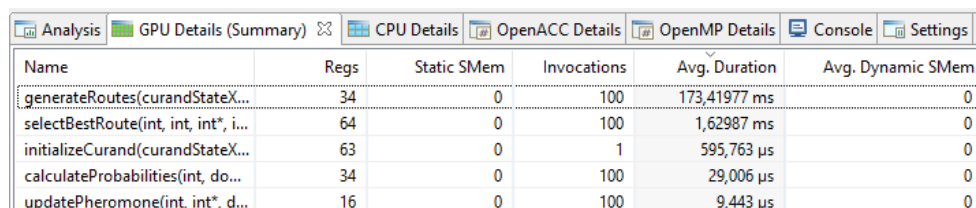
20. ábra: VRP futásidője GPU-n az iterációk függvényében (logaritmikus)

Ahogy az várható volt, a diagramon jól látszik, hogy az iterációk száma és a futási idő között lineáris összefüggés van. Kevés iteráció esetén a linearitást rontják a mérésbe beleszámolt memóriaműveletek, amik időigénye néhány (1-3) iteráció időigényével növelik a program futási idejét. Ebből kifolyólag a többi paraméteren végzett mérések során nincs jelentősége, hogy hány iterációval futtatjuk az algoritmust, ha elég nagy az iterációk száma. Ha több iterációval végezzük a mérést, csak a szükséges idő fog növekedni, a mérési pontosság csak elhanyagolható mértékben.

Könnyű belátni, hogy ez a paraméter az egyszálú megvalósítás esetén is hasonló módon befolyásolja a futási időt.

5.3.4 Kernelek időaránya

A CUDA programok futásának vizsgálatához az Nvidia cég több szoftvereszközt is fejlesztett. Ezek közül az Nvidia Visual Profiler (NVVP) használatával olyan futási idejű metrikákat, mint a felhasznált erőforrások, függvényhívási és visszatérési időbélyegek, hőmérséklet, valamint vizuális formában is meg tudja jeleníteni a futó GPU programokat. Az NVVP használatával lehetőségem volt kernelre lebontva megtekinteni a felhasznált regiszterek számát, a memóriahasználatot és az összesített futási időt is.



Name	Regs	Static SMem	Invocations	Avg. Duration	Avg. Dynamic SMem
generateRoutes(curandStateX...	34	0	100	173,41977 ms	0
selectBestRoute(int, int, int*, i...	64	0	100	1,62987 ms	0
initializeCurand(curandStateX...	63	0	1	595,763 µs	0
calculateProbabilities(int, do...	34	0	100	29,006 µs	0
updatePheromone(int, int*, d...	16	0	100	9,443 µs	0

21. ábra: Statisztika az egyes kernelek erőforráshasználatáról az NVVP programban

A statisztikából látszik, hogy a Taillard 75a adatkészleten a leghosszabb ideig futó kernel a hangyák szimulálásáért felelős függvény, ami az idő megközelítőleg 99%-ában volt aktív. A többi adatkészletnél is hasonló volt a kernelidők eloszlása, a nagyobb adatkészletek esetében az útgenerálás még nagyobb arányú időt tett ki.

Ebből azt a következtetést vontam le, hogy az algoritmusnak ezt a részét a jövőben érdemes tovább optimalizálni, mert csak így érhető el a teljes algoritmus gyorsulása.

Megfigyeltem továbbá, hogy csak az útvonalgenerálásért felelős kernelek közül a legelőször futó függvény tart legtovább. Ez az időtöbblet a futási konfiguráció és az adatkészlet függvényében 1-5% között mozgott a többi kernelhíváshoz mérten. Ennek valószínűleg az az oka, hogy amíg a feromonértékek a kezdeti állapotban vannak, addig

az egyes hangyák erősen eltérő útvonalakat járnak végig és ez sok divergens végrehajtási utat eredményez, ezzel növelve a futásidőt. A későbbi iterációk hangyái kevésbé hajlamosak eltérni a már megtalált addigi legjobb utaktól, ezért csökken a divergencia okozta futtatási overhead. Kiugró eltérést csak az első iterációban történő kernelhívásnál figyeltem meg.

5.3.5 Heurisztikus algoritmus pontossága

A dolgozatomban nem vizsgáltam külön azt, hogy az egyes hangolható paraméterek hogyan hatnak a kapott közelítő eredményekre, pontosabban arra, hogy az optimálisnál mennyivel hosszabb utat talál meg a heurisztikus algoritmus.

Az általam végzett néhány erre vonatkozó mérés alapján a közelítő eredmények az adatkészlettől és a paraméterektől függően az optimális útvonal hosszához képest 25-50%-kal rosszabb becslést adtak a jármű útvonaltervezési probléma esetén.

Mivel a fő célom a futásidő vizsgálata volt, ezért az α , β , ρ és Q paramétereket nem elemeztem külön, mert Katona Géza munkájából [3] kiindulva ezeknek (racionális határokon belül) várhatóan nem lenne szignifikáns hatása az időigényre a hangyakolónia algoritmus esetén, vagy ha mégis, akkor az egyformán érintené az egy és többszálú implementáció futását.

6 Összefoglalás

A munkám során első lépésként megterveztem és megvalósítottam egy ACO működési elvű algoritmust a VRP megoldására. A megoldás során szem előtt tartottam, hogy a program képes legyen nagyméretű, ritka, irányított gráffal reprezentált problémákat is kezelni.

Ennek érdekében megvizsgáltam a CRS gráf tárolási eljárás alkalmazhatóságát. Mivel ez a tárolási mód nem túl elterjedt, ezért készítettem egy konvertert, ami a sokkal egyszerűbb szomszédsági mátrix formátumból képes CRS formába hozni a gráfot.

Az idáig egyszálú programot először egyetlen összefüggő GPU kernelként implementáltam, azonban ennek a megközelítésnek a hátrányai miatt ezt a megoldást elvettem. Ehelyett felosztottam az algoritmust logikailag összetartozó részekre és ezeket külön párhuzamosítottam. Az egyes kerneleket ezután úgy alakítottam át, hogy a lehetőségekhez mérten leginkább kihasználják a grafikus kártya erőforrásait, vagyis a memória sáv szélességet és a számítási kapacitást. Később az optimalizációkat átalakítottam, hogy a kihasználtsági metrikák maximalizálása helyett a futási idő legyen minimális.

A párhuzamosítás során folyamatosan ellenőriztem a GPU algoritmus helyességét az egyszálú implementációt referenciaként használva. A kész algoritmust egy apró módosítással felkészítettem az utazóügynök probléma kezelésére. Ennek szintén ellenőriztem a helyességét.

Végül elkészítettem a futási idő méréséhez használt végleges kódokat, futtattam a méréseket, majd az eredményeket és megfigyeléseket a dolgozatomban rögzítettem.

A munkám során sok sokat tanultam az alacsony szintű programozási megoldásoktól kezdve egészen az absztrakt párhuzamos programozási módszerekig. Ez a terület felkeltette az érdeklődésemet. Sok kihívással szembesültem, de minden egyes újabb GPGPU koncepció megértése hatalmas sikerélményként ért. Korábban, azt tapasztaltam, hogy miután megtanultam valamilyen új módszer vagy technológia alkalmazását, az visszatekintve sokkal egyszerűbbnek tűnt. Ebben az esetben ez az érzés nincs meg bennem, habár az is igaz, hogy kezdő szinten sikerült csak elsajátítani még az ismereteket. A jövőben szeretnék jobban elmélyedni ebben a területben, de nem kizárólag a CUDA-ra korlátozottnak, hanem más technológiák is érdekelnek (pl. SYCL).

7 Továbbfejlesztési lehetőségek

Az itt bemutatott programok átláthatóbbá tehetők az általam kézzel implementált algoritmusok helyett elterjedt szoftverkönyvtárak használatával. A futási idő a GPU-n további módosítások alkalmazásával még tovább csökkenthető, az általam elvégzett átalakítások nem merítik ki az összes elérhető lehetőséget.

Kellően kis méretű feldolgozandó gráfok esetén a CRS tárolás során érdemes megfontolni az adatok padding-gel való kiegészítését a memória gyorsítótár jobb kihasználása érdekében.

A legjobb utat kiválasztó kernelt érdemes lehet párhuzamos redukcióval megvalósítani, mivel jobban kihasználja a GPU erőforrásait és várhatóan gyorsabb futást eredményez. Az útvonalkereső kernelben a relatív valószínűségek összegének meghatározásához szintén lehetne tovább optimalizálni párhuzamos redukció alkalmazásával. Ehhez érdemes megfontolni a dinamikus parallelizmus, vagyis a kernelből indított kernelhívás alkalmazását. Ekkor azonban figyelembe kell venni az egyszerre futtatott szálak számának növekedését.

A hangyakolónia algoritmus más problémák megoldására is használható. Az útvonalkereső kernel módosításával két pont közötti útkeresés és más gráfokon definiált feladat is megvalósítható. Ennek segítségével az algoritmussal nemcsak a TSP és VRP, hanem más, gráfként reprezentált problémák is megoldhatóvá válnak, például bizonyos munkaütemezési feladatok.

Bizonyos problémák esetén előnyösebb, ha a hangyák nem ugyanabból a csúcsból indulnak. Ennek alkalmazását is érdemes megvizsgálni, ahol azt a feladat jellege megengedi, mert összességében jobb közelítést eredményezhet.

Az alkalmazott heurisztikus megközelítés előnye, hogy az alapprobléma feltételeit az analitikus megoldásokkal ellentétben könnyű egyéb megszorításokkal kiegészíteni, például VRP esetén a járművek maximális számával. Ennek köszönhetően a hangyakolónia optimalizáció egy jól skálázható és problémák széles palettájára megoldást jelentő algoritmus.

8 Irodalomjegyzék

- [1] J. L. Hennessy és D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, Morgan Kaufmann.
- [2] M. Hayward, „Quantum Computing and Shor’s Algorithm,” 2015.
- [3] K. Géza, „Személyszállítási feladatra multimodális útvonaltervező és optimalizáló algoritmus fejlesztése,” 2019.
- [4] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. v. d. Vorst és editors, Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Philadelphia: SIAM, 2000.
- [5] R. S. Pierre L’Ecuyer, „TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators - User’s guide, compact version,” 2013. [Online]. Available: <http://www.iro.umontreal.ca/~simardr/testu01/guideshorttestu01.pdf>. [Hozzáférés dátuma: 09 12 2022].
- [6] M. Manssen, M. Weigel és A. K. Hartmann, „Random number generators for massively parallel simulations on GPU,” 2012.
- [7] NVIDIA Corporation, „CUDA C++ Programming Guide,” 2022. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. [Hozzáférés dátuma: 09 12 2022].
- [8] L. Zhang, M. Wahib és S. Matsuoka, „Understanding the Overheads of Launching CUDA Kernels,” 2019.
- [9] F. J. Diego, E. M. Gómez, M. Ortega-Mier és Á. García-Sánchez, „Parallel CUDA Architecture for Solving de VRP with ACO,” *Industrial Engineering: Innovative Networks. Springer London*, p. 385–393, 2012.
- [10] M. Harris, „CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops,” NVIDIA Corporation, 22 04 2013. [Online]. Available:

<https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/>. [Hozzáférés dátuma: 09 12 2022].

- [11] M. Harris, „How to Access Global Memory Efficiently in CUDA C/C++ Kernels,” NVIDIA Corporation, 07 01 2013. [Online]. Available: <https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/>. [Hozzáférés dátuma: 09 12 2022].
- [12] É. Taillard, „Vehicle routing instances,” 06 07 2022. [Online]. Available: <http://mistic.heig-vd.ch/taillard/problemes.dir/vrp.dir/vrp.html>. [Hozzáférés dátuma: 09 12 2022].