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

Herédi Péter

PÁRHOZAMOS ÚTVONALTERVEZÉS A SZÁMÍTÁSOK KISZERVEZÉSÉVEL

KONZULENS

Dr. Dudás Ákos

BUDAPEST, 2015

Tartalomjegyzék

Összefoglaló

A közelmúltban a számítógépek, majd ezt követően a mobil eszközök számítási teljesítménye rohamos tempóban növekedésnek indult, a többmagos és többprocesszoros rendszerek bevezetésének köszönhetően. Azonban a bennük szunnyadó teljesítmény kihasználásához a korábbi programozási és szoftverfejlesztési nézetek nem alkalmasak. A korábbi, úgynevezett szekvenciális programok képtelenek kihasználni a többmagos architektúrákban rejlő potenciált. Helyüket átveszik az úgynevezett párhuzamos, vagy más néven több szálú alkalmazások.

Dolgozatomban szeretném bemutatni, hogy mekkora többlet teljesítmény nyerhető a párhuzamos programozás technikáinak alkalmazásával mobil és szerver környezetben. További célom egy határérték meghatározása, amit átlépve már megéri kiszervezni a számításokat mobil környezetből a szerverek/felhő világába. Ennek szemléletes bemutatása érdekében választottam kutatási témámul az útvonaltervezést, mivel manapság a legtöbbünk zsebében ott lapuló okos telefonok mindennapi életünk részévé tették az erre épülő alkalmazásokat.

Célom egy olyan teszt szoftver elkészítése, amin szemléltethető a különböző útkereső gráf algoritmusok - mind szekvenciális mind párhuzamos megvalósítással – nyújtotta teljesítmény. Jelen dolgozatban a kiválasztott algoritmusok felhasználásával kísérletet teszek arra, hogy meghatározzam azt a küszöb értéket, ami alatt a számítást a mobil eszköz, felette pedig a nagyobb számítási kapacitással rendelkező szerverek végzik. A vizsgálatokat olyan szempontok alapján végzem el, mint a futási idő, fogyasztás, hálózati adatátvitel költsége.

Abstract

Recently, the computing power of computers and mobile devices started in a fast paced growth, thanks to the development of multi-core and multi-processor systems. However, the previously used programming and software development methods are not suitable to exploit these systems' latent powers. The former, usually known as sequential programs are unable to exploit the potential of multi-core architectures. So their place is taking over by the so called parallel, also known as multi-threaded applications.

In my study, I would like to show the performance gains that can be obtained by parallel programming techniques in mobile and server environment. My further goal is to identify a limit which determines whether to do the calculations on the mobile device or it would be more efficient to offload the computation into server or cloud environment. For this reason to well represent these gains, I chose the field of path planning as the topic of my study, because nowadays most of us have a smart phone in their pockets so we can get in touch with applications that use some kind of path finding in our everyday life.

My goal is creating a test application, which can illustrate the different path finding graph algorithms – both sequential and parallel implementations – provided performances. In this study I attempt to determine a threshold - with the help of selected algorithms - which divides the computational works into two groups. Problems in the first group (under the threshold) should be calculated on the mobile devices, but the second group demands more computational power so it could be more efficient to do the necessary calculations on servers. To determine this threshold I will take into account the running times of the algorithms, the power consumption of the devices and the cost of network communication.

1 Bevezetés

Mostanra elmondható, hogy az okos telefonok használata mindennapjaink szerves részét képezik. Olyan funkciók kerültek bele ezekbe a kisméretű hordozható eszközökbe, mint például helymeghatározás (GPS, cella információ, Wi-Fi alapú), kamera, nagyméretű érintőkijelző, különbféle szenzorok (gyorsulásmérő, fény érzékelő, giroszkóp).

A termék paletta közel 100%-át három nagy platform foglalja el, ezek részesedésük sorrendjében: Android, iOS, Windows Phone. Sok hasonlóságot hordoznak magukban funkcionalitás és felépítés tekintetében. A hardver architektúra [1], amelyre épülnek közös ARM gyökerekkel rendelkezik és mostanra minden platform fel van készítve a több magos processzorok kihasználására. Sok szempontból ezek a rendszerek kiválthatják a hagyományos asztali számítógépeket és laptopokat. Az elérhető alkalmazások száma folyamatosan növekszik, és elképesztő sokszínűséget mutat.

Egy hasonlóképpen felkapott téma az utóbbi időben a felhő alapú rendszerek terjedése (kiváltva a hagyományos szerverek üzemeltetését). Egyre több szolgáltató kínál különböző szintű szolgáltatásokat ezeken a rendszereken és teszi ezeket elérhetővé bérelhető formában. Ezeknek a rendszereknek a lényege, hogy nincs a fizikai rendszer a fejlesztő/felhasználó kezében, csupán távoli erőforrásokat kap (pl.: virtuális gép), aminek nem ismeri se a pontos elhelyezkedését, se azt, hogy milyen hardveren fut. Előnye, hogy rugalmasan igényelhető további szükséges erőforrás és tetszőlegesen beköthető a meglevő rendszerünkbe.

1.1 Cél

A dolgozatom célja, hogy megvizsgáljam egy alsó-közép kategóriás okos telefon képességeit és teljesítményét, valamint összevessem a nagy teljesítményű szerverekkel, olyan szempontból, hogy mikor melyik eszközt érdemes a feldolgozáshoz választani, ebben az esetben az útvonal kereséshez.

Ezt a technikát az irodalom „computation offloading” néven emlegeti. Lényege, hogy egy kisebb számítási kapacitással rendelkező eszközt mentesítünk a feladata egy része alól, kiszervezve a szükséges számításokat egy nagyobb teljesítményű infrastruktúrába (szerver, cloud, grid). Mobil környezetben való használatával a hordozható eszköz üzemideje megnőhet, az alkalmazás futás ideje pedig csökkenhet, mivel a számításokat nem a saját processzora végzi, neki már csak a kommunikációval és a megjelenítéssel kell foglalkoznia. Ennek a megosztásnak az optimális határát szeretném meghatározni a dolgozat végére.

A válasz eléréséhez méréseket végzek az elkészített mobil és szerver alkalmazások felhasználásával. Figyelembe fogom venni a feladat lefutási idejét, a processzor és memória használatot, a hálózati kommunikáció mértékét és mobil készülékek esetében a fogyasztást.

Ezen eredményekből kívánok létrehozni mérőszámokat, amik segítségével kijelölhetek egy határt az elvégzendő feladat nagyságára és komplexitására vonatkozóan. Ezen határ alatt érdemesebb lesz a mobil készüléken elvégezni a számításigényes feladatokat és e fölött már érdemesebb a szerver felé fordulni és vállalni a hálózati kommunikáció szükségességét.

1.2 Dolgozat felépítése

A dolgozat következő fejezetében bemutatom, hogy melyik platformra és hozzá kapcsolódó technológiákra esett a választásom a mérések elvégzéséhez.

A harmadik fejezetben feltárom, hogy milyen egyéb kutatások folytak vagy folynak jelenleg is a világban a párhuzamos programozás és a mobil környezetben való feladatok kiszervezésének témájában. Ezekből milyen következtetések vonhatók le és milyen hasznos tudást tudok belőlük meríteni és felhasználni jelen írásomban.

A negyedik fejezetben röviden bemutatom az algoritmusokat és módszereket, amikre a két alkalmazásom épül, hogy ezek megismerésével a mérési eredmények könnyebben értelmezhetőek legyenek.

Ezt követően az ötödik fejezetben részletesen bemutatom az elkészített mobil alkalmazást és szerver oldali web szolgáltatást. Részletezem a felépítésüket és az elkészítésük során hozott fejlesztői döntéseket. Ezen felül bemutatom, hogyan lehet a szerver oldali kódot egyszerűen, akár egy otthoni számítógépen elérhetővé tenni a hálózaton.

Az utolsó előtti fejezetben első lépésként bemutatom a mérésekhez használt gráfokat, ezek tulajdonságaiból lehet következtetni a feladatok komplexitására, ami a következtetések levonásához szükséges lesz. Ezt követően sorra veszem a teszt rendszert alkotó fizikai elemeket, mint a szerver gép és a mobil telefon tulajdonságait. Szintén ebben a fejezetben fognak helyet foglalni a mérési eredmények és ezek önmagában valamint összevetésben való elemzése.

Az utolsó fejezetben pedig megvizsgálom, hogy milyen további érdekes irányokba lehetne további vizsgálatokat folytatni a szerverek és mobil eszközök közötti munka elosztás és kommunikáció területén.

2 Felhasznált eszközök

Ebben a fejezetben azokat az eszközöket fogom tárgyalni, amelyeket felhasználtam a rendszerek összeállításához és a mérések elkészítéséhez.

2.1 Platformok kiválasztása

Amikor nekiálltam a dolgozat elkészítésének abból indultam ki, hogy a korábban elkészített útvonal tervező algoritmusaimat [2] fogom felhasználni a jelen vizsgálataimhoz. Ezzel a döntésemmel el is köteleztem magam a Microsoft .NET [3] platform mellett, legalábbis a szerver oldali komponenst illetően ez tűnt a legkézenfekvőbb megoldásnak, a régebbi .NET-es kódok integrációja miatt. Így döntöttem végül a .NET keretrendszer 4.5.2-es verziója és a Windows Communication Foundation (továbbiakban WCF) mellett. A WCF megadta nekem azt a lehetőséget, hogy a már elkészült kód köré könnyen és rugalmasan építhessek web szolgáltatást.

Ami a mobil platformot illeti, ha csak azt az irányt vizsgáljuk, hogy hogyan működhet együtt a mobil eszköz és a szerver, akkor nem kellett volna hasonló megkötéssel számolnom. Bármely vezető platform (Android, iOS, Windows Phone) alkalmassá tehető lett volna a szerverrel való együttműködés kialakítására.

Azonban célom volt a kizárólag okos telefonon való futás vizsgálata is, ugyan csak a már korábban elkészült kódok felhasználásával, így végül a Windows Phone operációs rendszer legújabb végleges, 8.1-es verziója mellett döntöttem. Ezt használó készülék a rendelkezésemre is állt.

Ezen platformok választása azt is megmutatja, hogyan emelhető át kód mobil platformról szerver oldalra, illetve fordítva a hatékonyabb feldolgozás érdekében a kód jelentősebb módosítása nélkül. További előnye az egy gyártótól származó platformok használatánál, hogy elegendő volt egy fejlesztői környezetet konfigurálnom.

2.2 Fejlesztői környezet

A fejlesztés Microsoft Windows 8.1 operációs rendszeren, a Visual Studio 2013-as változatát használva folyt. Ez lehetővé tette, hogy egy eszközből kezeljem a szükséges projekteket. Ennek segítségével egyszerűen konfigurálhattam a WCF szolgáltatásomat és jól használható szerkesztőt nyújt a Windows Phone alkalmazás felületének megtervezéséhez.

Továbbá mobil fejlesztésnél lehetőséget nyújtott rá, hogy a kódot egyenesen a telefonon is debugolhattam, vagy használható a beépített emulátor is.

Ezen felül a mérések elkészítésénél sokat használtam a beépített analizátor eszközt a Performance Analyzer-t. Ezzel az eszközzel használat közben végezhettem méréseket az alkalmazásokon. Rákapcsolva az eszközt olyan információkhoz lehet jutni, mint a CPU használat mértéke, a memória kihasználtsága, a hálózati kommunikáció mértéke, vagy az alkalmazás energiafogyasztásának nyomon követése. Nagyon hasznosnak bizonyult, hogy ezek az eszközök közvetlenül a fejlesztő környezetben rendelkezésemre álltak és nem igényeltek külön konfigurációt.

3 Irodalomkutatás

A témában folytatott kutatásom során sok olyan írással találkoztam, amelyek gráf algoritmusokkal vagy útvonaltervezéssel foglalkoznak. Ahogy a publikációk készítési ideje közeledik napjainkhoz, egyre jobban áttérnek a több magos rendszerek kihasználását elősegítő párhuzamos megvalósítások taglalására.

Ami az évek során azonban nem változott az a kutatás alapja. A Dijkstra és az A* algoritmus a szakma által leginkább felkapott algoritmusok, ezek különbféle plusz optimalizációval ellátott változatairól több kutatás is folyt.

Wiktor Jakubiuk és Keshav Puranmalka 2011-es, „*Parallelization of Dijkstra's Algorithm: Comparison of Various Priority Queues*" [7] című munkájában a Dijkstra algoritmus megvalósításához különböző adatstruktúrákat tesztel. Priority Queue különböző megvalósításait (Fibonacci Heap, Binomial Heap, Relaxed Heap) alkalmazzák az algoritmushoz, amelyben a magok számának megfelelően *n* részre osztják a sort és szinkronban lépve külön magokon végzik a részek feldolgozását. A méréseiket különböző méretű és sűrűségű gráfokon végezték és megállapították, hogy a Fibonacci és Relaxed Heap-et használó változatok jelentős gyorsulást érnek el. A modern hardverek által kínált masszív párhuzamosság egyre jelentősebbé teszi a párhuzamos Dijkstra algoritmus optimalizálását. Ennek a publikációnak az eredményei vezettek el engem is oda, hogy vizsgálódásomhoz használjak Priority Queue-t is használó algoritmust.

Az MIT-n (Massachusetts Institute of Technology) Kevin Kelley és Tao B. Schardl által készített „*Parallel Single-Source Shortest Paths"* [8] című munkában a kutatók a Dijkstra algoritmust alkalmatlannak tartják a masszív párhuzamosításra, mivel a megvalósításban Priority Queue-ra támaszkodik és ennek az adatszerkezetnek a módosítása a teljes struktúra átrendezésével is járhat, így minden iterációban több elem is kiértékelődik, nem csak a sor elején álló. Ehelyett a Gabow algoritmust ajánlják. A valós úthálózatokon készített méréseik azonban azt mutatják, hogy a Dijkstra algoritmus felülmúlja a párhuzamos Gabow algoritmus eredményeit.

Jad Nohra és Alex J. Champandard, az Intel munkatársai által „*The Secrets of Parallel Pathfinding on Modern Computer Hardware"* [9] címen publikált műben az

A* algoritmus skálázódását vizsgálják meg többmagos környezetben, mivel a videó játékokban használt mesterséges intelligencia megvalósításhoz a fejlesztők előszeretettel alkalmazzák. A megvalósításhoz az Intel TBB könyvtárát használták. Méréseik alapján az A* algoritmus magja olyan masszív számításokat hajt végre, hogy a párhuzamosítás adminisztrációs költsége eltörpül a több végrehajtó szál használatának előnye mellett. Eredményeik szerint az A* algoritmus jobban skálázódik, mint a legtöbb másik algoritmus, így szinte minden egyes új szál és processzor bevezetése után gyorsult a végrehajtás. Továbbá megállapították, hogy a mai hardver architektúrákon a vizsgálat terének (gráf mérete) és komplexitásának csökkentése sokkal nagyobb gyorsulást eredményez, mint az adatstruktúra memória igényének a csökkentése, az alkalmazott cache hozzáférési ideje nem rontja a teljesítményt.

Mint látható, a gráf algoritmusokon alapuló útkeresés napjainkban is aktuális témának számít, főleg az egyre terjedőbb többmagos hardverek miatt. Az ezekben rejlő erőt a régi, jól bevált algoritmusok átalakításával és kiegészítésével képesek vagyunk kihasználni. A kutatások abba az irányba mennek, hogy hogyan lehet a lehető legtöbb felesleges munkát megspórolni, miközben a lehető legtöbb magot vonjuk be a számításokba. Ennek megvalósítására különböző hierarchikus adatstruktúrákat vezetnek be, különbféle módszerekkel partícionálják a keresési teret, hogy a magok egymástól független adatokkal dolgozhassanak, és minél ritkábban kelljen kommunikálniuk egymással.

Ezek az írások mind nagy gépes környezetekről szólnak, azonban az egyre fejlődő okos telefonok is hasonló több magos architektúrára alapoznak, így megindultak a kutatások ezek bevonására a számításokba.

Karthik Kumar, Jibang Liu, Yung-Hsiang Lu és Bharat Bhargava 2012-es „*A Survey of Computation Offloading for Mobile Systems"* [10] című írásukban nagyszerűen összefoglalják a „computation offloading" mibenlétét, a jótékony hatásait és a technikákat, amik ezt lehetővé teszik. Kitérnek rá, hogy a 2000-es évek elejétől kezdve vált technológiailag megvalósíthatóvá és elkezdődhetett a konkrét alkalmazási területek kialakítása. Olyan technológiák kerülnek kihasználásra, mint a virtualizáció, nagy sávszélességű hálózatok, web szolgáltatások és felhő rendszerek. Kétféle irányzatot különböztetnek meg a számítások szétosztására: a statikus, amikor előre megmondjuk, hogy mi lesz kiszervezve és a dinamikus, amikor algoritmusok figyelik a környezeti változókat és ennek megfelelően döntenek, hogy mit és mikor adjanak át a

szervereknek. Várakozásaik szerint a jövőben továbbra is fontos lesz a számítások kiszervezése a megfelelő eszközökre, különösen az egyre nagyobb teret nyerő mobil eszközök körében és az ezekből szerveződő hálózatok ellátásában.

A georgiai egyetem kutatói fejlesztették ki a COSMOS („*Computation Offloading as a Service for Mobile Devices*”) [11] rendszert, amely mint egy szolgáltatás segíti a mobil eszközök esetében a „computation offloading” menedzselését. A kutatók szerint a legnagyobb probléma a témakörben, hogy a mobil eszközökön véletlenszerűen növekszik meg a számítási kapacitásra való igény, a felhő szolgáltatók azonban csak lassan tudnak reagálni a bővítési igényekre és általában egy fix időkeretig mindenképpen meg kell tartani a foglalt erőforrást, még ha nem is használjuk azt. Tehát egy növelési igény bejelentése nagy többlet költséggel jár. Ennek kezelésére alakították ki a COSMOS rendszert, amely dinamikusan végzi a foglalásokat és az újabb beérkező igényeket igyekszik a már foglalt erőforrásokra allokálni, így minél jobban kihasználva azokat mielőtt újabb foglalási igényt delegálni a felhő szolgáltató felé. A kész rendszerrel bebizonyították, hogy csökkenthetők a járulékos költségek, és képes megfelelően kezelni a különböző hálózati kapcsolatokat.

Afnan Fahim, Abderrahmen Mtibaa, és Khaled A. Harras Carnegie Mellon Egyetemen a fentiektől eltérő felosztáson gondolkodtak. *„Making the Case For Computational Offloadingin Mobile Device Clouds”* [12] című munkájukban bemutatják a mobil felhők koncepcióját, amiben a mobil eszközök nem a „nagy gépeknek” adják át a szükséges számításokat gyorsítás és energia hatékonyság céljából, hanem a közelükben található, a mobil felhő hálózatához csatlakozó más mobil eszközöknek. Az elképzelés teszteléséhez külön API-t (kiszervezési döntések elvégzésére) és Android alkalmazást készítettek, majd különböző környezeteket szimulálva végezték a tesztjeiket. Kimutatták, hogy 50%-kal csökkennek a futási idők és 26%-kal a fogyasztás egy így felépített hálózatban futtatva a feldolgozást, mintha csak a lokális eszközön hajtódna végre.

A publikációkból látható, hogy érdemes foglalkozni a kérdéssel, aktív kutatási területnek számít a világban. Azonban az is leszűrhető, hogy nincsen abszolút, mindig a legjobb teljesítményt nyújtó megoldás, ha kicsit más szempontból közelítjük meg a témát már más eredményekre juthatunk az eddig sikerrel alkalmazott technikákkal is.

4 Algoritmusok

Ebben a fejezetben röviden össze szeretném foglalni, hogy milyen általam korábban készített algoritmusokra [2] alapozom a méréseimet.

Két jól ismert legrövidebb utat kereső algoritmust valósítottam meg 4-4 verzióban, ezek pedig a Dijkstra és az A* [4] algoritmusok. Elkészítettem szekvenciális és párhuzamosított változataikat, beépített adatstruktúrák (SortedList) és saját készítésű tárak (PriorityQueue) és rendező algoritmusok (MergeSort) segítségével.

Az elkészítésük során a párhuzamos feldolgozás nyújtotta lehetőségek kihasználása volt a célom, hogy a hardver erőforrások jobb kihasználásával sebesség növekedést érjek el. A felhasznált adatszerkezeteknél pedig a konkurens hozzáférés jó kezelése mellett figyeltem rá, hogy memória használat szempontjából is jól skálázódjanak.

4.1 Gráfot tároló adatstruktúra

A kedvező memória használat érdekében egy olyan adatstruktúrát alkalmaztam, amely egyesíti a szomszédossági mátrixok és az él listás tárolás előnyös tulajdonságait. Az implementációmban a Dictionary osztályt használom, amelyen függvényeken keresztül különböző viselkedési formákat valósítottam meg. Nem kellett figyelembe vennem, hogy több szálas környezetben kerül felhasználásra, mivel csak olvasás történik az adatstruktúrában, módosítás nem.
Az adatstruktúra szerkezete:

```
Dictionary<int, Dictionary<int, int>>
```

A szótár kulcsa jelképezheti a gráf egyes csúcsait, valamint gondolhatunk rá úgy is, mint a szomszédossági mátrix egy sora. Ezzel címzünk egy újabb szótárat, melyben, ha szomszédossági mátrixként gondolunk rá, akkor a kulcs megadja, hogy az adott sorban melyik oszlopban található az értékként megadott súly. Nem tárolom el bele azokat az eseteket, amikor nincs két csúcs között él, hanem egy property és egy függvény segítségével, ha olyan elemre hivatkozik a program, ami nincs benne a szótárban, akkor a végtelennek megfelelő értéket adja vissza. Így kiküszöböltem azt a problémát, hogy ritka gráfoknál nagy helyet foglal az elérhetetlen csúcsok tárolása.

Elem eléréséhez használt függvény:

```
public int GetAt(int row, int col)
{
    Dictionary<int, int> cols;
    if (graph.TryGetValue(row, out cols))
    {
        int value = int.MaxValue;
        if (cols.TryGetValue(col, out value))
        {
            return value;
        }
    }
    return int.MaxValue;
}
```

Él listás megközelítésnél pedig a belső szótárba egyszerűen felveszem a szomszédos csúcs-súly párokat, és ha kíváncsi vagyok, hogy milyen élek mennek ki egy csúcsból, csak kulcsként kell használnom a forrás csúcsot és ki kell olvasni a megcímzett szótárból a kulcsokhoz tartozó értéket, hála a szótár asszociatív címzésű működésének ez egy gyors művelet.

4.2 Naiv szekvenciális Dijkstra algoritmus

Első felhasznált algoritmus olyan megoldás, ami az alábbi pszeudo kódot követi:

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
        dist[v]  := infinity ;
        previous[v]  := undefined ;
    end for
    dist[source]  := 0 ;
    Q := the set of all nodes in Graph ;
    while Q is not empty:
        u := extract_min(Q)
        if dist[u] = infinity:
            break ;
        end if
        for each neighbor v of u:
            alt := dist[u] + length(u, v);
            if alt < dist[v]:
                dist[v]  := alt ;
                previous[v]  := u ;
            end if
        end for
    end while
    return dist[], previous[];
end function
```

1. ábra Dijkstra pszeudokód

A kódban a fentebb leírt gráf tároló adatszerkezeten túl használok további két szótárat (distances, previous) a távolságok és az előző elemek nyomon követésére, továbbá egy listát (nodes) ami kezdetben az összes csúcsot tartalmazza. Analóg módon a pszeudo kóddal, a megvalósításom első körben inicializálja a távolságok és az utakat tároló szótárakat, a kezdőpont kivételével végtelenre állítva a távolságok értékét és felépítem a csúcsokat tároló listát.

Ezt követően a fő ciklusban addig megyek, amíg van csúcs a nodes listában vagy a legkisebb távolságra levő csúcs a keresett csúcs. A legkisebb távolságú elem kinyerését a nodes listán hívott `nodes.Sort((x, y) => distances[x] - distances[y])` beépített rendező függvény és lambda kifejezéssel átadott rendezési feltétel segítségével oldottam meg.

Ez az implementáció azonban nem hatékony mivel minden egyes ciklusban újra rendezi a teljes listát és a szótárakból való adat kiolvasással tölti a futásidő nagy részét.

4.3 Naiv párhuzamos Dijkstra algoritmus

Az előző algoritmus implementációmban megállapítható volt, hogy a beépített rendezés sokszori futtatása jelenti a szűk keresztmetszetet, ezért ezt lecseréltem egy saját, párhuzamos Merge sort implementációra.

A Merge sort algoritmus egy oszd meg és uralkodj elven működő *O(n*log(n))* komplexitású összehasonlításokon alapuló rendező algoritmus.
Az algoritmus két részből áll:

1. Felbontja a rendezetlen tömböt *n* darab altömbre egész addig, amíg mindben már csak 1 elem lesz, ez már rendezettnek tekinthető.
2. Ismétlődően végrehajtva a merge lépést az altömbökön, ezzel létrehozva új rendezett tömbrészeket, egészen addig, amíg már csak 1 tömb marad, ez lesz a végeredmény.

A párhuzamos megvalósításhoz a .NET-ben található TPL-t (Task Parallel Library) használtam, annak Parallel.Invoke() metódusának segítségével a résztömbökön futó számításokat hozzárendeltem külön szálakhoz, így partícionálva a feladatot, azonban végül a merge funkció lefutása szekvenciálisan történik meg, így nem érhető el teljes kihasználtság.

A saját rendező algoritmuson túl további optimalizációként, ahol lehetséges volt lecseréltem a szótárakat tömbökre, valamint az alternatív legrövidebb utak keresését egy Parallel.ForEach() ciklusban végzem el. Ennek következtében szálbiztos osztályokat

(ConcurrentDictionary, ConcurrentBag) kellett használnom a konkurens hozzáférés miatt, ez némiképpen lassítja a hozzáférést. Így sikerült nagymértékű sebesség növekedést elérnem, de továbbra is a rendezés a szűk keresztmetszet.

4.4 Okos szekvenciális Dijkstra algoritmus

Újra gondoltam az algoritmus működését és áttértem szomszédossági mátrixos megvalósításra, kihasználva, hogy az elkészített gráf adatstruktúrámmal ez nem jár memória többlettel. Továbbá teljesen elvetettem a kiegészítő szótárak használatát, helyettük tömböket használok, ezzel megszabadulva jelentős visszafogó erőtől (cache barátiabb adatstruktúra).

A rendezés helyett minimumkeresést valósítottam meg, ami futása közben nyilvántartja az éppen aktuális legkisebb értéket és a hozzá tartozó indexet a csúcsok tömbjében. Az algoritmus addig fut, amíg a legközelebbinek ítélt csúcs indexe meg nem egyezik a keresett csúcséval, vagy minden továbblépési lehetőség végtelen értéket ad vissza. Ezekkel a módosításokkal sikerült jelentős sebesség növekedést elérnem a korábbi párhuzamos implementációval szemben is.

4.5 Okos párhuzamos Dijkstra algoritmus

A közös erőforrás használat miatt a következőket változtattam a szekvenciális implementációhoz képest. A cél elérésének detektálásához használt változókból saját példányt kapnak az egyes szálak, és a `Parallel.For(0, n, (i, loopState)` függvényben a loopState segítségével leállítok minden szálat, ha bármelyik megtalálta a keresettet idő közben. Ennek a megvalósításnak a segítségével a minimum távolságokat és az alternatív utakat szálanként más-más irányba képes keresni, míg meg nem találja az optimálist. Ezzel az implementációval elértem azt a szintet, hogy a gráf adatstruktúrából való kiolvasás adja a futási idő legnagyobb részét.

4.6 Szekvenciális A* algoritmus

Az A* algoritmus a legismertebb megvalósítása a legjobbat-először keresésnek. Ez az algoritmus egy informált keresési stratégia - ellentétben a Dijkstra algoritmussal – ami a plusz információt a felhasznált heurisztika függvény segítségével viszi bele a kiértékelő függvénybe, ami alapján végül eldől az útválasztás.

Ennek megvalósítására a következő pszeudokódot követtem:

A-Star(Graph graph, Node start, Node goal, HeuristicFunction h)
1. O← make_priority_queue(startNode) // open list
2. C←make_hash_table // closed list
3. While O not empty loop
    1. n ← O.remove_front() //O is sorted by *f(n)=g(n)+h(n)* values
    2. If goal (n) return n
    3. If n is found on C → continue
    4. //otherwise
    5. S ← successors (n)
    6. For each node s in S
        1. Set s.g=n.g+w(n,s)
        2. Set s.parent=n //for path extraction
        3. Set s.h=h(s) //to calculate *f*
        4. O←s
    7. C←n
4. Return null //no goal found

2. ábra A* pszeudokód

Létrehoztam két listát az algoritmus működéséhez, a nyílt és a zárt listát. A nyílt lista tárolja a még meg nem látogatott csúcsokat, a zárt lista pedig azokat a csúcsokat tartalmazza, amiket az algoritmus már kiválasztott és kifejtett, végül ebből kerül összeállításra a legrövidebb út.

A nyílt listát a SortedList osztállyal valósítottam meg, melyben a rendezés a már megtett út és a heurisztika összege alapján történik. A heurisztikát két pont légvonalbeli távolsága adja meg, amit a beépített GeoCoordinate osztály segítségével határozok meg. Mivel a zárt listában nem lényeg a rendezettség, ezért az egy rendezetlen List-ként kerül megvalósításra.

4.7 Párhuzamos A* algoritmus

Elképzeléseim szerint az algoritmus futása során az a lépés gyorsítható jelentősen a párhuzamosítással, amikor egy csúcson állva megvizsgálja, hogy melyik szomszédos csúcsra érdemes tovább lépni. Ezért ezt a részt futtatom egy Parallel.Foreach ciklusban.

Pár helyen változtatnom kellett a vezérlési szerkezeteimen, mivel ellentétben a nyelvi elemként szereplő foreach-csel, a Parallel.Foreach minden elemre meghívja a törzsében definiált függvényt, így a következő iterációra való ugrást nem a continue, hanem a return utasítás kényszeríti ki.

Mivel a zárt listát csak szekvenciális kódrészekben módosítom, párhuzamosan csak olvasom, így nincs szükség szinkronizáció bevezetésére az esetében. Ezzel ellentétben a nyílt listát írom a párhuzamosított kódszakaszon, ezért mielőtt hozzáadnék a listához, zárolom a listát a lock utasítással, majd a lock belsejében még egyszer ellenőrzöm a belépési feltételt, nehogy előálljon az a helyzet, hogy időközben egy másik szál már módosította a listát, majd hozzáadom az elemet.

Várakozásaim szerint ezek a módosítások csak abban az esetben hoznak gyorsulást, ha a csúcsnak elegendően sok szomszédja van, tehát elég adatot kell feldolgozni, hogy a gyorsulás meghaladja az adminisztrációs költségeket, különben a párhuzamosítás csak hátrányt okoz.

4.8 Szekvenciális A* Priority Queue adatszerkezettel

A szekvenciális algoritmus tovább gyorsítására a nyílt lista megvalósítását cseréltem le egy saját implementációjú Priority Queue-ra. A prioritási sor tulajdonsága, hogy az elemeket önmagában egy prioritás szerint rendezve tárolja és elem kivételekor a legnagyobb prioritásút adja vissza, majd törli a sorból. A belső adatszerkezet megvalósítására kupacot használtam.

A kupac tulajdonság azt követeli meg, hogy az apa csúcs értéke mindig kisebb legyen a fiú csúcs értékénél. A kupac belső működése mindig fent tartja ezt az állapotot új elem beszúrása és minimális elem törlése után újra rendezi a csúcsokat. A belső rendezettség miatt a legközelebbi következő csúcs kiválasztása csak a gyökérben levő elem kivételét jelenti. Ez a művelet hatékonyan elvégezhető, komplexitása $O(\log_2 n)$,

ahol n a csúcsok száma. Továbbá új elem beszúrása is $O(\log_2 n)$ komplexitású művelet és a kupac újra rendezésével jár.

Az algoritmus működését tekintve nem volt szükség nagy változtatásra, a nyílt listát lecseréltem a prioritási sorra és értelemszerűen alkalmaztam a függvényeit a `SortedList` függvényei helyett.

4.9 Párhuzamos A* Priority Queue adatszerkezettel

Elkészítettem a Priority Queue-t használó implementáció párhuzamosítását az előző párhuzamos verzióval analóg módon a Parallel.Foreach bevezetésével a szomszédok vizsgálatakor. Az elkészített prioritási sorom sajnos nem szál biztos ezért a használata közben figyelnem kell a megfelelő szinkronizációs primitívek használatára. Erre a feladatra egy egyszerű lock-ot használtam a kritikus szakaszokon, mint a sorhoz való hozzáadás, a sor hosszának lekérdezése és a tartalmazási vizsgálat.

4.10 Vizsgálati megfontolások

A jelenlegi felhasználásukban érdemesnek tartottam megvizsgálni, hogy ezek az algoritmusok hogyan skálázódnak a mobil készülékeken, összehasonlítva a szerver oldali megvalósítással. Bár a .NET keretrendszert használja mindkét platform, az elérhető szolgáltatásokban mégis akad különbség. Ezen felül eltér a hardver architektúra is, az asztali és szerver gépek terén az x86/x64 a domináns, még a hordozható készülékekben egyértelműen az ARM számít piac vezetőnek.

Tehát érdekesnek tartottam megvizsgálni, hogy az asztali környezetben jól és egyszerűen használható TPL hogyan skálázódik a mobil architektúrákon. Vajon a kisebb hardveres erőforrások mellett is képes lesz a task alapú végrehajtással gyorsítani a feladatok lefutásán? Meg fogom vizsgálni, hogy a hardver nagyobb mértékű kihasználása milyen viszonyban van a rendszer fogyasztásával, a gyorsabb lefutás vajon ellensúlyozza-e a nagyobb aktivitást?

5 Fejlesztett komponensek

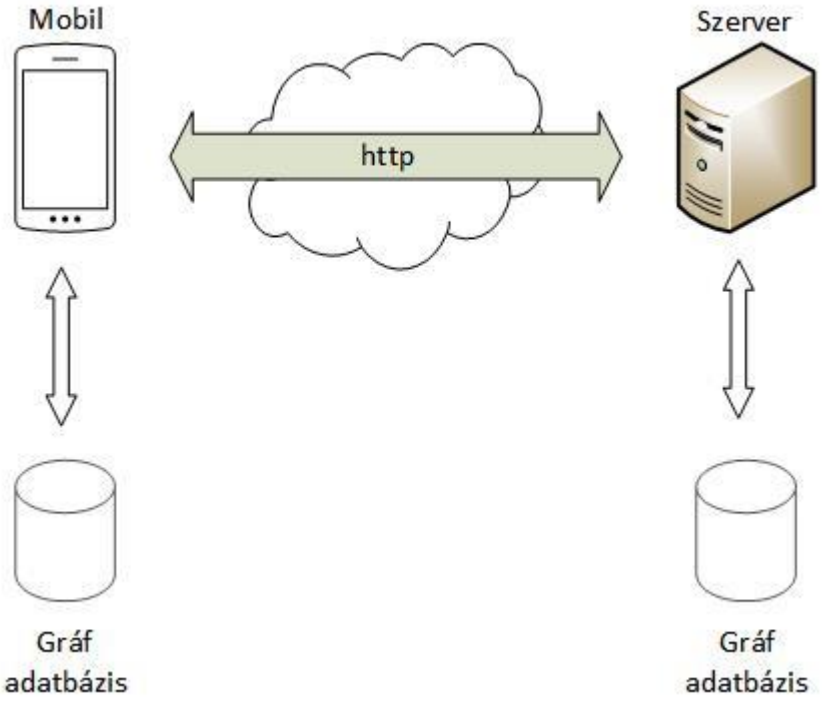
Jelen fejezetben bemutatásra kerülnek az elkészített komponensek és ezek kapcsolódásának módja. Bemutatom milyen technikákat alkalmaztam az elkészítésük során és ezektől milyen eredményeket várok.

5.1 A rendszer architektúrája

A mérési környezet az alábbi esetet szimulálja:

- az útvonalhoz szükséges térkép adott, mobil kliens esetén a mobil eszközön, szerver oldali feldolgozás esetén a szerveren
- egy tervezési feladathoz a térkép (gráf), a kezdeti és végpont azonosítójának megadása szükséges
- a felhasználó megadja a fentebb vázolt adatokat, majd elindítja a számítást a kiválasztott eszközön
- ezen számítás költségét mérhetjük a rendszer segítségével

A mérési környezet architektúrája a következő:

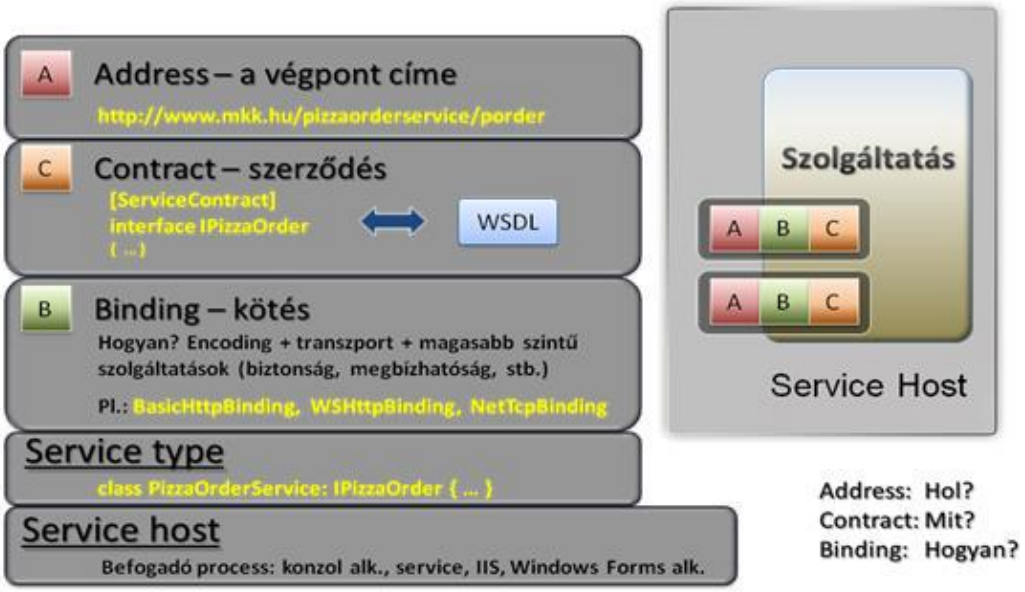


3. ábra Rendszer architektúra

5.2 WCF alapú szerver komponens

A WCF [5] egy szolgáltatásorientált és elosztott alkalmazások fejlesztését lehetővé tevő újgenerációs technológia, amely a .NET egyik legnagyobb erősségét jelenti. A keretrendszer 3.0-ás verziója óta érhető el és egységes modellt képvisel a korábbi széttagolt ASMX alapú web szolgáltatásokat, .NET Remoting-ot, COM+-t, stb. használó technikákkal szemben. De fontos megjegyezni, hogy ezek használatát nem zárja ki, képes velük együtt működni, így a visszafele kompatibilitás is biztosított. Ezen felül megoldott az interoperabilitás más platformok felé is (Java, WS* szabványok).

Lehetőséget nyújt szinkron és aszinkron kommunikáció megvalósítására. A szolgáltatásban definiálható egy vagy több, különböző tulajdonságokkal rendelkező végpont (endpoint), majd ezeket kiajánlva rajtuk keresztül kommunikálhatnak a kliensek a szolgáltatással. Egy endpoint architektúrális szempontból három dolgot fog össze: Address (szolgáltatás címe), Binding (felhasznált protokoll), Contract (elérhető műveletek).



4. ábra Végpont felépítése

Az én rendszerem központjában is egy ilyen szolgáltatás található, ami magában foglalja a gráfokat tartalmazó erőforrások elérését és kezelését, a beérkező kéréseknek megfelelően meghatározza a legrövidebb utat a kért algoritmussal, a kért gráfon, majd válaszként elküldi a kliensnek a számítás eredményét. Ez a kérés-válasz kapcsolat kétféleképpen érhető el jelenleg.

Létrehoztam egy „hagyományos” SOAP alapú, főként szinkron kommunikációra kialakított interfészt, amelyre elsősorban asztali alkalmazások és web oldalak kapcsolódhatnak. Ennek elkészítésére létrehoztam egy interfészt, amit majd a szolgáltatást nyújtó osztályom megvalósít. Ez olyan metódusokat tartalmaz, mint a rendszer példányának inicializálása, gráf letöltése kliens oldalra, legrövidebb út keresése. Ezt az interfészt el kellett látnom a `[ServiceContract]` attribútummal, ezzel jelezve, hogy ő egy web szolgáltatás. A benne található metódusok azonban nem lesznek automatikusan elérhetők, amelyeket szeretném, hogy kívülről is el lehessen érni, el kell látni az `[OperationContract]` attribútummal.

A binding-ot, ami megadja a kommunikációs csatorna felépítését végül egy egyszerű httpBinding-gal oldottam meg, amit a WCF beépítve tartalmaz BasicHttpBinding néven. Így meghagytam a lehetőségét, hogy a későbbiekben ne csak .NET platformról legyen használható a szolgáltatás.

A SOAP alapú szolgáltatásnál az API nyújtotta DataContract alapú sorosítást használtam (ha nagyobb szabadságra van igény használható az XmlSerializer), így a megfelelő osztályokat ellátva a `[DataContract]` attribútummal, valamint a propertyket és tagváltozókat pedig a `[DataMember]` attribútummal, a keretrendszer automatikusan kezelte a sorosítást a kommunikáció során.

Ezt követően a szolgáltatást felhasználó kliens alkalmazásokban a publikált WSDL leíró fájl alapján generálhatók a szükséges interfészek és üzleti entitások, vagy ha a mi kezünkben van a kliens és szerver fejlesztése is, akkor a Visual Studio-val Service Reference-ként hozzá adva a projekthez a fejlesztőeszköz létrehozza a szükséges entitásokat és konfigurációs állományokat. Későbbiekben, ha a szolgáltatás interfésze nem módosul, csak az elérési címe akkor az alkalmazás újra fordítása nélkül a web.config fájlban módosítható a kívánt cím. További beállításokra is lehetőséget ad ilyen formában, mint például a maximális csomag mérete és az időtúllépés idejének beállítása.

Külön végpontként definiáltam a mobil kliensek felé nyújtott Representational State Transfer (REST) alapú szolgáltatás interfészt. Erre azért volt szükség, mert a hordozható eszközök világában nem elfogadható a nagy sávszélesség igénnyel rendelkező, szinkron működésre kihegyezett SOAP alapú rendszerek használata. Ehelyett HTTP üzenetek formájában, HTTP metódusokon keresztül történik a kommunikáció. Használható műveletek a következők: Get, Put, Post és a Delete. Az

egyes erőforrások pedig URL-ek segítségével címezhetők. A korlátozott hálózati elérés miatt a leggyakoribb adat formátum a Json, ez a SOAP által használt Xml-nél jóval kisebb méretű üzeneteket eredményez, így ilyen formában továbbítódnak az üzenetek a HTTP csomagokban.

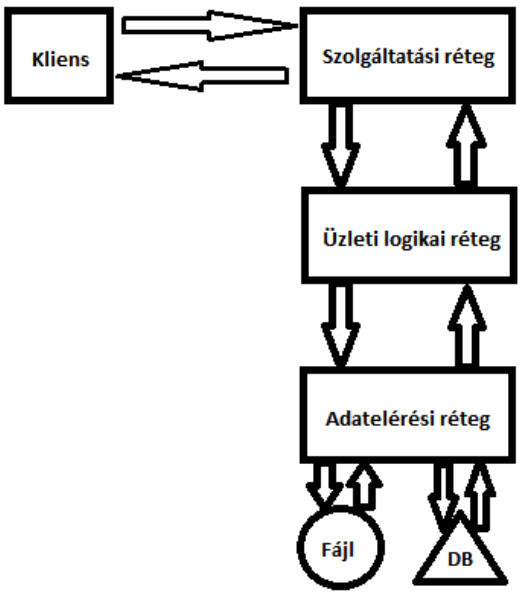
A WCF a REST alapú működést is támogatja a WebAPI-n keresztül, hasonlóképpen, mint az előbb bemutatott attribútumokat alkalmazó SOAP megvalósítás esetében.

Nem kellett új interfészt létrehoznom, használhattam a korábbi SOAP alapú szolgáltatás interfészét, csupán a kiajánlott metódusokat kellett kiegészítenem néhány attribútummal. Mivel a felhasznált Request és Response objektumaim is összetett objektumok, így a Post metódus használata mellett döntöttem minden funkcióm esetében.

Ennek használata a következő: a kiajánlott szolgáltatás műveleteit elláttam a `[WebInvoke]` attribútummal, ez jelzi a HTTP üzenetek kezelését. Ezen belül meg kellett adni, hogy milyen HTTP metódust kezel az adott függvény (esetemben mindenhol Post, Get esetén `[WebGet]` használandó), milyen ki és bemenő adatformátumot vár (esetemben Json) valamint, hogy a bázis címtől számítva milyen URL-lel érhető el. Ennyi változtatásra volt csupán szükség a SOAP-os kódhoz képest, mivel az ott fel attribútumozott osztályokat a keretrendszer képes Json formátumban sorosítani. A web.config fájlba pedig fel lett véve egy új endpoint, amelynek a címén a REST alkalmazás figyel.

A szerver alkalmazásomat rétegezett architektúrában készítettem el, a rétegeket pedig interfészek mentén kapcsoltam össze. Ettől a megközelítéstől azt az eredményt várom, hogy a rétegek anélkül legyenek cserélhetők, hogy a rendszer többi komponensének működésében változtatást vonnának maguk után.

Ennek fényében a jelenlegi adatelérési rétegben a gráfokat Json formátumú szöveges fájlokból olvassa fel a rendszer, de könnyedén lecserélhető lenne egy adatbázis alapú működésre, ameddig a felsőbb réteg felé ugyanazokat az entitásokat adja tovább.



5. ábra Rétegezett architektúra

Az adatelérési réteg fölött elhelyezkedő üzleti logikai réteg adja az alkalmazás magját, ez az a komponens, ami a korábban készült gráf algoritmusok kódját tartalmazza. Ebben a komponensben is szabadon változtatható a működés, építhetők be új algoritmusok, ezek konkrétan nincsenek kivezetve a szolgáltatási rétegbe, a kéréssel beérkező Enum határozza meg, hogy milyen algoritmust kell futtatni, ez az entitás szorul bővítésre. A legfelső, szolgáltatási réteg tartja a kapcsolatot a külvilággal, ide esnek be a kliensek kérései, majd hív tovább a megfelelő üzleti logikai függvénybe. A kliensek kódja egészen addig nem igényel változtatást, amíg a szolgáltatási réteg által nyújtott interfész nem változik.

5.2.1 ASP.NET Web Forms alapú web kliens

A szerver oldali szolgáltatás fejlesztése közben szükségem volt egy kliensre, amellyel tesztelhettem az elkészült funkciókat. Erre a szerepre a SoapUI nevű alkalmazás is használható lett volna, ez az eszköz lehetővé teszi, hogy a szolgáltatás WSDL-je alapján legenerálja a SOAP üzeneteket a műveletekhez, majd ebben a megfelelő helyeket kitöltve elküldhető a kívánt kérés a szerver felé, a visszakapott választ pedig szintén Xml formátumban jeleníti meg.

Ennél azonban jóval szemléletesebbnek találtam azt a megoldást, hogy ASP.NET felhasználásával készítsek egy weboldalt, ami kliensként funkcionál és kapcsolódik a web szolgáltatásomhoz. Service Reference-ként Visual Studio-ból kötöttem bele a szolgáltatást, így automatikusan legenerálódtak a kommunikációhoz

szükséges entitások és egy proxy osztály, amin keresztül egyszerűen hívhatóvá váltak a kiajánlott műveletek.

Minden szükséges műveletet és beállítási lehetőséget kivezettem a felületre, hogy a funkciók könnyedén elérhetők legyenek tesztelés és mérés során. Az ASPX weboldal felelős tehát a kommunikáció lebonyolításáért és a megjelenítésért, a számítási kapacitást a szerver szolgáltatja.

5.2.2 Hosztolás IIS alól

Ahhoz, hogy a web szolgáltatást úgymond éles környezetben tesztelhessem nem tűnt célravezetőnek, hogy például egy konzol alkalmazásból hosztolva tegyem elérhetővé más komponensek számára. Ehelyett az IIS web szerverre való telepítés mellett döntöttem, ehhez szükséges egy svc fájl felvétele a projektbe minden szolgáltatáshoz. Fordítás és telepítés után minden futáshoz szükséges erőforrás a web szerverre került, nem volt többet szükség Visual Studio-ra a használatához, a konfigurációs beállításokat nem kódban valósítottam meg, hanem a web.config fájlban, így ennek módosítása után a következő beérkező hívást a szerver már az új beállítások alapján kezeli.

További előnye az IIS használatának, hogy on-demand működést tesz lehetővé, tehát üzenet érkezésekor aktiválódik, ha azonban nincs terhelés lekapcsolhatja magát. A management konzoljából pedig probléma esetén újra lehet indítani egyes szolgáltatásokat vagy az egész szervert.

Az ASPX kliens is erre a web szerverre került telepítésre, de éles körülmények között ezek tipikusan másik szerverre kerültek volna, a jobb terhelés elosztás érdekében (mind hálózati forgalom, mind processzor idő tekintetében).

5.3 Windows Phone 8.1

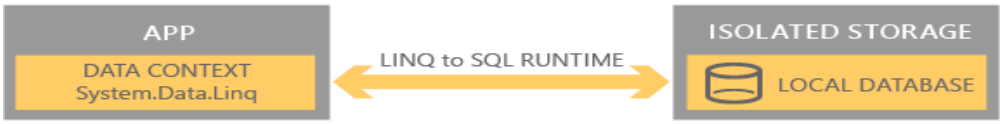
A Windows Phone [6] operációs rendszer a Microsoft okos telefonokra szánt rendszere, amit a 8-as verzióra teljesen átdolgoztak és megkapta a WinRT API-t a régi win32 helyére. Ezzel sok felesleges funkciótól megszabadult a rendszer és áttérhettek egy teljesen COM alapú, ebből következőleg programozási nyelv független API-ra. Ezzel párhuzamosan a Windows Phone-ra elérhető .NET keretrendszer is változott, sok mobil alkalmazások számára felesleges funkció kikerült belőle, vagy átalakításra került és közvetlenül a WinRT-s megvalósítást hívja egy egyszerű csomagoláson keresztül.

Erre a rendszerre készítettem el az útvonal tervező alkalmazásom két különböző változatát: a helyben futót és a szerver számítási kapacitását igénybe vevőt. Két különböző működésről beszélhetünk, azonban csak egy alkalmazás készült hozzá, felületről vezérelhető, hogy melyik működési elvet használjuk. Ha az eredmények igazolják a szerver létjogosultságát, akkor ez később akár automatikus átkapcsolásra is cserélhető, az elérhető internet hozzáférés függvényében.

5.3.1 Kliens oldali feldolgozás

Az első bemutatásra kerülő verzió teljes mértékben kliens oldali feldolgozást használ, tehát semmilyen formában nem fordul a szerver felé. Ahogyan azt a szerver alkalmazásnál is bemutattam, már meglévő kód köré építem az alkalmazásokat, ebben az esetben se történt másként, azonban számos változtatással kellett élnem, ahogy azt lentebb részletesen bemutatom. Ez az alkalmazás azért készült el, hogy megvizsgálhassam, hogy mire képes egy alsó kategóriás telefon önmagában. Mérési eredményeket a következő fejezetben közlök.

Meg kellett oldanom, hogy minden futáshoz szükséges erőforrás elérhető legyen a készüléken, beleértve a gráfot tároló adatstruktúrák. Két lehetséges megoldás közül választhattam, használhatok lokális adatbázist, ami a telefonon található úgynevezett Isolated Storage területen tárolja az adatbázis fájljait és LINQ to SQL-en keresztül írható és olvasható az alkalmazásból.

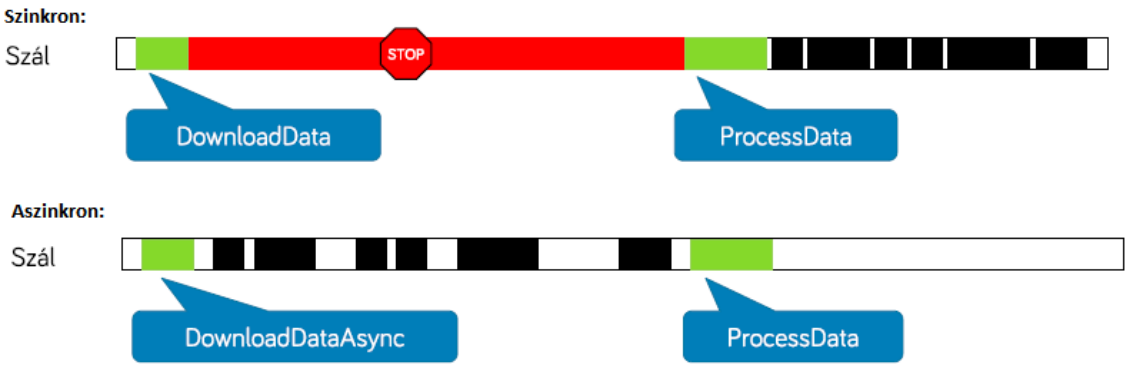


6. ábra Lokális adatbázis

Forrás: https://msdn.microsoft.com/en-us/library/windows/apps/hh202860%28v=vs.105%29.aspx

Másik lehetőség, hogy a Visual Studio projektben erőforrásként/content-ként felveszem a szerveren is használt szöveges fájlokat és deploy-kor ezek is felmásolódnak a telefonra. Végül a második megoldás mellett döntöttem és miután szembesültem vele, hogy az elérhető .NET verzió nem kínál fájl kezelést, a WinRT nyújtotta fájl kezelési lehetőségeket és a Json.NET Json konvertert használtam fel a szükséges gráfok memóriába töltéséhez.

Ezen a ponton szembesültem a mobil platform „másságával" első körben. Ez pedig nem mást jelentett, mint az aszinkron műveletek preferálása a szinkron végrehajtással szemben. Erre egyfelől az alkalmazás reszponzivitásának megőrzése szempontjából van szükség, valamint az erőforrások jobb kihasználásához.



7. ábra Szinkron vs Aszinkron

Aszinkron végrehajtás során a rendszer a kérés elküldése és a válasz megérkezése között más feladatokat is végezhet és amikor a válasz megérkezett visszatér annak feldolgozására. Erre az értesítésre korábban callback függvényeket és esemény kezelőket kellett készteni. Azonban a C# nyelv most már tartalmaz egy konstrukciót az aszinkronitás kód barát kezelésére, ez az úgynevezett async-await minta. Az async módosító szóval jelzett metódusok futhatnak aszinkron módon, hozzájuk a fordító egy állapotgépet generál, hogy visszatéréskor onnan folytathassa a feldolgozást ahol abbamaradt. Az await kulcsszút pedig egy async metóduson belül lehet használni, ez jelzi, hogy átadhatja a futás jogát, amíg az aszinkron Task le nem fut.

A fájl kezelő async metódusomban például a következő kódsorok végzik a gráfot tartalmazó fájl felolvasását és a memóriában használt struktúrává konvertálását:

```
StorageFolder folder = await Windows.ApplicationModel.Package.Current
                      .InstalledLocation.GetFolderAsync("Resources");
StorageFile file = await folder.GetFileAsync(source.ToString() + ".txt");
IList<string> json = await Windows.Storage.FileIO.ReadLinesAsync(file);
Dictionary<int, Dictionary<int, int>>graph = JsonConvert.
                                        DeserializeObject<Dictionary<int, Dictionary<int, int>>>
                                        (json[0]);
```

A StorageFolder és StorageFile is WinRT interfészeket valósítanak meg, ezek használatával érhetők el a telefon belső tárhelyén található könyvtárak és fájlok, a .NET nem nyújt plusz lehetőségeket a fájlkezeléshez.

További korlátozásokkal is számolnom kellett, ami a .NET tudását illeti. Nem érhető el a korábbi SortedList adatstruktúra, amit az egyik algoritmusom felhasznál, így át kellett alakítanom kicsit a működését és SortedDictionary került beépítésre a helyére. További érdekes eltérést fedeztem fel a Windows Phone 8.1-es verzióján és az asztali operációs rendszeren elérhető .NET keretrendszer lokációkkal kapcsolatos API-jában. Még a Windows rendszereken elérhető a GeoCoordinate osztály, igen bő eszköztárral, köztük a lehetőséggel, hogy két koordinátáival adott pont közötti távolságot képes meghatározni, addig a Windows Phone legújabb verziójából ez kikerült. Mivel az A* algoritmusaim pont ezt használták heurisztikaként, kénytelen voltam egy kerülő utat keresni és implementálni a Haversine formulát, amit a GeoCoordinate is használt. Furcsa döntésnek tartom, hogy egy mobil eszköznél, ahol többféle helymeghatározási lehetőség is létezik, csonkításra került az ezt kezelő API.

5.3.2 Szerver bevonása a feldolgozásba

A másik elkészített működési módban az alkalmazás a web szolgáltatás REST alapú végpontjára küldi a kéréseket, ahelyett, hogy a telefon maga dolgozná fel a felhasználói felületről érkező parancsokat. Ettől a működéstől azt várom, hogy a végrehajtás a hálózati kommunikáció ellenére is gyorsul, mindeközben a telefon fogyasztása mérséklődik, mivel csak a megjelenítést és a kommunikációt kell kezelnie.

Ahogy már korábban leírtam, a REST a működése során HTTP kérés-válasz párokat használ, ezek kezelését kellett megoldanom a kliens oldalon. Az üzenetek kezelésére készítettem egy generikus metódust, amit meghívásakor megfelelően paraméterezve elvégzi a memóriabeli objektum Json string-gé alakítását. Továbbá összeállítja és elküldi a megadott URL-re a http kérést, majd amikor megérkezik a

válasz, feldolgozza annak tartalmát és visszaalakítja üzleti entitássá a kapott Json üzenetet. A hálózati kommunikáció miatt aszinkron végrehajtás használandó, ennek megfelelően ismét az async-await szerkezetet használtam.

A függvény szerkezete a következő:

```
public async Task<K> SendMessage<T,K>(T request, string uriString)
{
    ulong? reqSize = 0;
    ulong? resSize = 0;
    using (var client = new Windows.Web.Http.HttpClient())
    {
        var uri = new Uri(uriString);
        Windows.Web.Http.HttpStringContent input = new Windows.Web.Http.HttpStringContent(
                                                   JsonConvert.SerializeObject(request),
                                                   Windows.Storage.Streams.UnicodeEncoding.Utf8,
                                                   "application/json");
        var res = await client.PostAsync(uri, input);
        res.EnsureSuccessStatusCode();
        resSize = res.Content.Headers.ContentLength;
        reqSize = res.RequestMessage.Content.Headers.ContentLength;
        string responseBody = await res.Content.ReadAsStringAsync();
        messageSize += (reqSize.Value + resSize.Value);
        tbMessageSize.Text = messageSize.ToString();
        tbEllapsedMilliseconds.Text = sw.ElapsedMilliseconds.ToString();
        return JsonConvert.DeserializeObject<K>(responseBody);
    }
}
```

Mint látható, ebben a függvényben végzem az üzenetek méretének a mérését is.

Egy érdekes hibával szembesültem a fejlesztés során, mégpedig, hogy a .NET nyújtotta HttpClient, bármilyen Timeout paramétert is állítok be neki figyelmen kívül hagyja azt és 1 perc elteltével timeout miatt leáll a feldolgozás. Ezért tértem át a WinRT nyújtotta HttpClient alkalmazására, ennél nem találkoztam ilyen problémával, idomult a szerver oldalon beállított időtúllépés értékéhez. A szerver oldal Post metódussal várja az üzeneteket, ezt a PostAsync metódus biztosítja számomra, amiben elküldi az előtte összeállított kérés üzenetet.

6 Mérési környezet és mérési eredmények

Ebben a fejezetben méréseket fogok végrehajtani az elkészült alkalmazásokon, amelyek során kitérek az elért futási időkre, a processzorok eltérő terhelési szintjeire valamint a telefon fogyasztására és a hálózati kommunikáció mértékére. Majd ezekből az eredményekből egy összesítést készítve kívánom meghozni a döntést, hogy mi az a határ, amikor már érdemes bevonni a szerver oldalt is a munkába.

Először azonban vegyük szemügyre milyen gráfokon és milyen hardverekkel készülnek a mérési eredmények.

6.1 Gráfok szerkezete

Mérésekhez két csoportba osztottam a gráfokat, az első csoportba tartoznak a véletlen generált közepes sűrűségű gráfok. Sűrűség alatt azt értem, hogy egy csúcsnak átlagosan hány szomszédja van. Korábbi vizsgálataimból [2] úgy vettem észre, hogy az algoritmusok viselkedésének különbségeit az 1-5000 közötti szomszédok száma jól mutatja, ezért ilyen gráfokat használtam most is 10, 100, 1000, 10 ezer, 100 ezer csúcsszámmal.

Másik csoportba tartozik az a gráf, amit a BKK menetrend adatbázisa alapján generáltam.[1] Ez jelenti számomra a valós körülmények legpontosabb modellezését. Ebben az esetben a gráfnak 5453 csúcsa van és egy csúcs átlagosan 30 szomszéddal rendelkezik.

[1] http://www.bkk.hu/tomegkozlekedes/fejlesztoknek/

6.2 Fizika rendszer elemek

A hardver eszközök tekintetében két gyökeresen eltérő rendszerről beszélhetünk. Először lássuk a szerverként használt számítógép felépítését:

- Processzor: AMD FX-8320, 8 mag, 3.5GHz – 4.0GHz
- Architektúra: x86-64
- Memória: 8GB DRR3-1600
- Operációs rendszer: Windows 8.1

Nokia Lumia 520:

- Processzor: 2 mag, 1GHz
- Architektúra: ARMv7
- Memória: 512MB LPDDR2
- Operációs rendszer: Windows Phone 8.1

Látható tehát, hogy a két hardver komponens közel sincs egymáshoz nyers számítási kapacitás terén. Lássuk, hogyan nyilvánul ez meg a mérési eredmények terén.

6.3 Futási idő

A futási időket a szerveren és a mobilon is Stopwatch osztály segítségével mértem.

6.3.1 Kliens lokálisan

A mérés az alábbi lépéseket foglalja magába:

1. adatok adottak, gráf betöltését nem mérem
2. felhasználó megnyomja az indító gombot, stopper indítása
3. algoritmus lefut és eredmény előáll, megjelenítés idejét nem mérem
4. vége, stopper leállítása

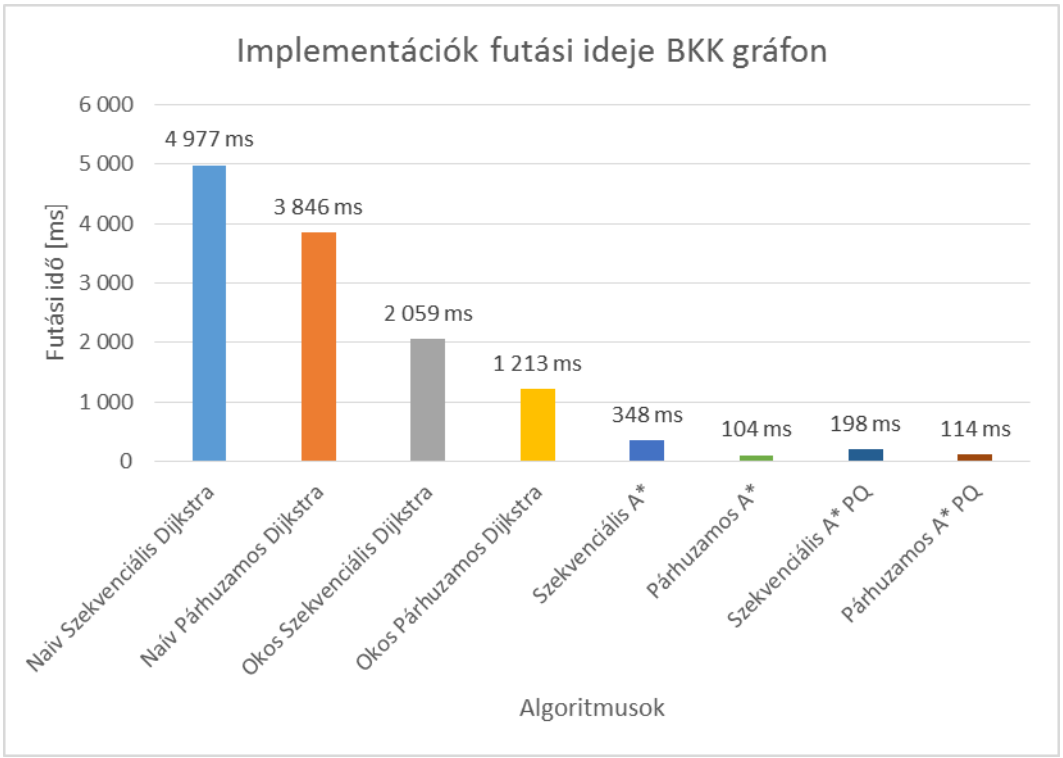
Ezt a lépés sorozatot ismételtem meg 10-szer, majd a kapott értékeket átlagoltam és felfelé kerekítettem.

Mérési eredmények:

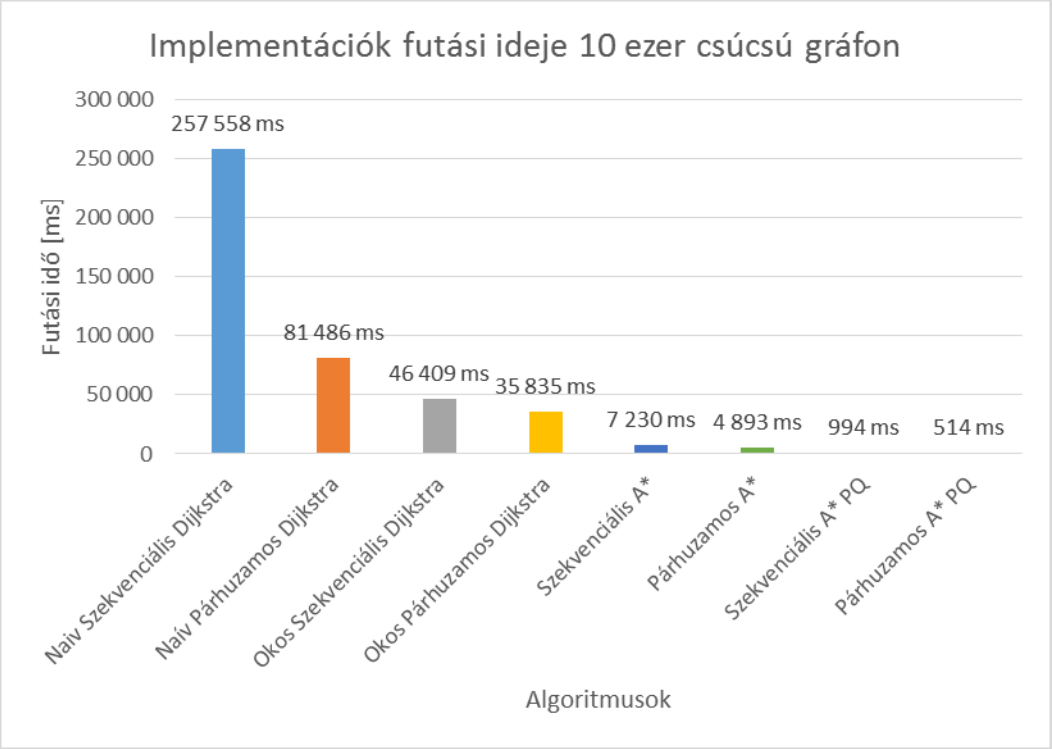
Mobil lokákis	Naiv Szekvenciális Dijkstra	Naív Párhuzamos Dijkstra	Okos Szekvenciális	Okos Párhuzamos Dijkstra
10 csúcs	31 ms	557 ms	9 ms	22 ms
100 csúcs	63 ms	361 ms	2 ms	80 ms
1000 csúcs	1 895 ms	2 742 ms	412 ms	705 ms
10000 csúcs	257 558 ms	81 486 ms	46 409 ms	35 835 ms
BKK	4 977 ms	3 846 ms	2 059 ms	1 213 ms

Mobil lokákis	Szekvenciális A*	Párhuzamos A*	Szekvenciális A* PQ	Párhuzamos A* PQ
10 csúcs	261 ms	148 ms	114 ms	18 ms
100 csúcs	6 ms	82 ms	12 ms	6 ms
1000 csúcs	263 ms	102 ms	12 ms	20 ms
10000 csúcs	7 230 ms	4 893 ms	994 ms	514 ms
BKK	348 ms	104 ms	198 ms	114 ms

8. ábra Futási idő mérési eredmények



9. ábra Futási idő BKK gráfon



10. ábra Futási idő 10 ezer csúcsú gráfon

Értékelés:

A telefonon lokálisan futtatva a 100 ezer csúcsú gráfot nem tudta a rendszer betölteni az 512MB memóriába, túllépte annak méretét. Véleményem szerint mobil környezetben, ha megfelelően jelezzük a felhasználónak, hogy nem állt le a működés, akkor 4 másodperc körüli várakozási idő még elfogadható. Így azt mondanám, hogy egy valós, mindennapi feladat esetén, ami esetemben a BKK menetrendből készített gráf volt, különösebb probléma nélkül tervezhetők gyors lefutású algoritmusok, de egyszerű megvalósításokkal is elfogadható eredmény érhető el.

A legkomplexebb probléma, amit sikerült kezelnem mobil készüléken a 10 ezer csúcsú gráf feldolgozása volt. Azonban ezen a szinten már a legegyszerűbb megvalósítások csődöt mondanak, komolyabb figyelmet igényel az algoritmusok megtervezése. Esetemben, csak a TPL-lel párhuzamosított, így 2 magot kihasználó A* algoritmus (és a tovább optimalizált változatok) hozta az éppen elfogadható szintet.

6.3.2 Kliens – szerver

A mérés az alábbi lépéseket foglalja magába:

1. adatok adottak, gráf a szerveren betöltve
2. felhasználó megnyomja az indító gombot, stopper indítása
3. várakozás a szerver oldali lefutásra
4. válasz feldolgozása, megjelenítés idejét nem mérem
5. vége, stopper leállítása

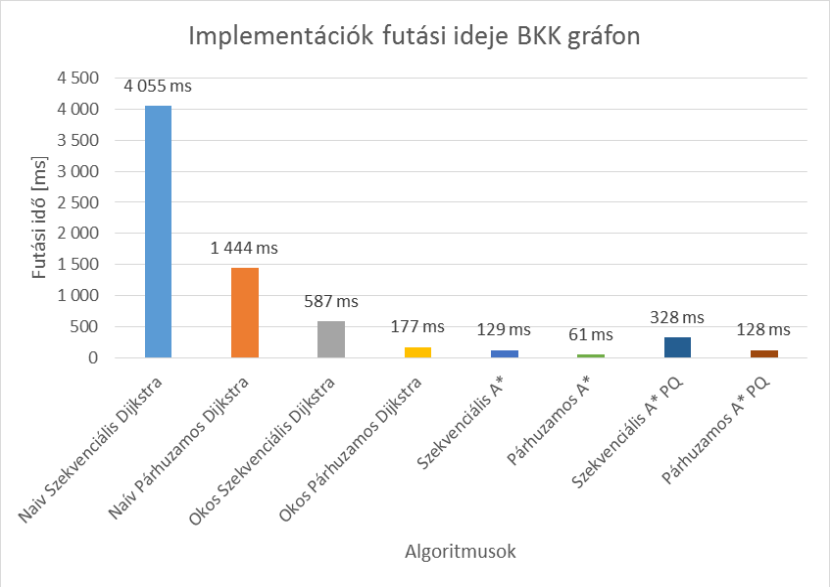
Ezt a lépés sorozatot ismételtem meg 10-szer, majd a kapott értékeket átlagoltam és felfelé kerekítettem.

Mérési eredmények:

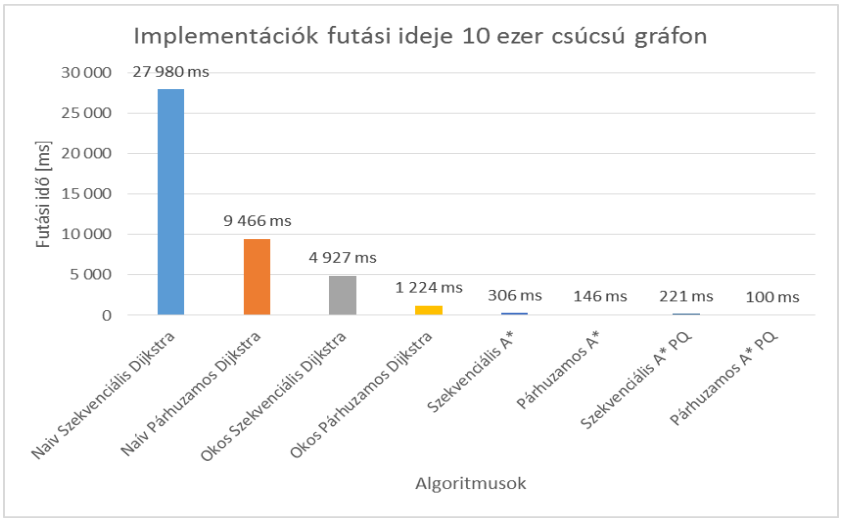
Kliens - szerver	Naiv Szekvenciális Dijkstra	Naív Párhuzamos Dijkstra	Okos Szekvenciális Dijkstra	Okos Párhuzamos Dijkstra
10 csúcs	29 ms	126 ms	29 ms	43 ms
100 csúcs	23 ms	30 ms	19 ms	20 ms
1000 csúcs	219 ms	175 ms	74 ms	46 ms
10000 csúcs	27 980 ms	9 466 ms	4 927 ms	1 224 ms
100000 csúcs	3 930 994 ms	1 286 083 ms	757 828 ms	156 201 ms
BKK	4 055 ms	1 444 ms	587 ms	177 ms

Kliens - szerver	Szekvenciális A*	Párhuzamos A*	Szekvenciális A* PQ	Párhuzamos A* PQ
10 csúcs	79 ms	35 ms	45 ms	41 ms
100 csúcs	24 ms	22 ms	23 ms	17 ms
1000 csúcs	22 ms	24 ms	27 ms	22 ms
10000 csúcs	306 ms	146 ms	221 ms	100 ms
100000 csúcs	28 454 ms	9 142 ms	27 711 ms	5 295 ms
BKK	129 ms	61 ms	328 ms	128 ms

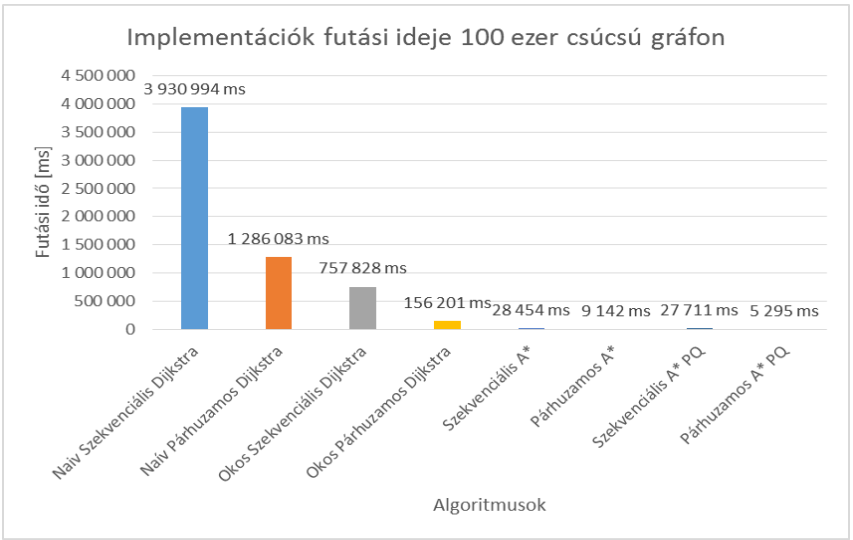
11. ábra Futási idő mérési eredmények



12. ábra Futási idő BKK gráfon



13. ábra Futási idő 10 ezer csúcsú gráfon



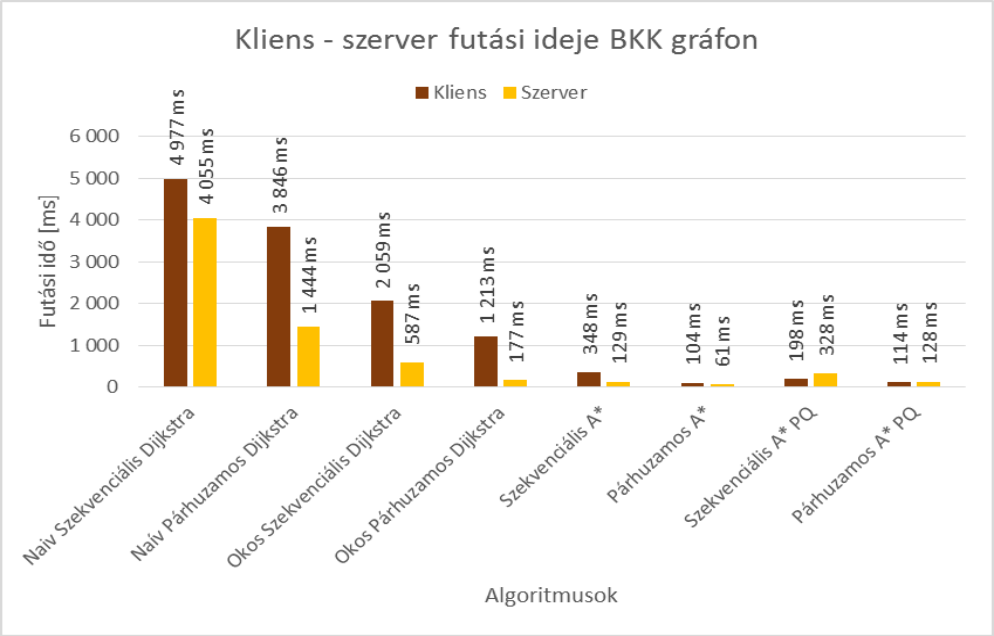
14. ábra Futási idő 100 ezer csúcsú gráfon

Értékelés:

A mérési eredmények tartalmazzák a hálózati kommunikáció idejét, így a szerver helyétől függően az értékek átlagosan 100-150ms-mal eltérhetnek. Megvizsgálva a hétköznapi problémát modellező BKK gráfos eredményeket, látható, hogy egy erős szerver gond nélkül megbirkózik ekkora problémával. 10 ezer csúcsú gráfnál, már a szerver se tudta minden algoritmussal tartani a 4 másodperc körüli eredményt, de a jobban átgondolt algoritmusok az okos párhuzamos Dijkstrától kezdve már jó eredményeket hoztak.

A 100 ezer csúcsú gráffal szemléltetett feladat már az asztali gépnek is problémát okozott. A legjobban optimalizált algoritmus sem tudta elérni az elfogadható futási időt. Ha mindenképpen ilyen komplexitású feladatot akarunk mobil eszköz felhasználásával futtatni, vagy felügyelni, akkor a megfelelő optimalizálás mellett vállalható futás idő kapható, főleg ha összehasonlítjuk az egyszerűbb algoritmusok lehangoló futási idejével.

Érdekességképpen megjegyezném, hogy a BKK gráfon összehasonlítva a kliens és a szerver teljesítményét, meglepő látni, hogy nagyon közel vannak egymáshoz.



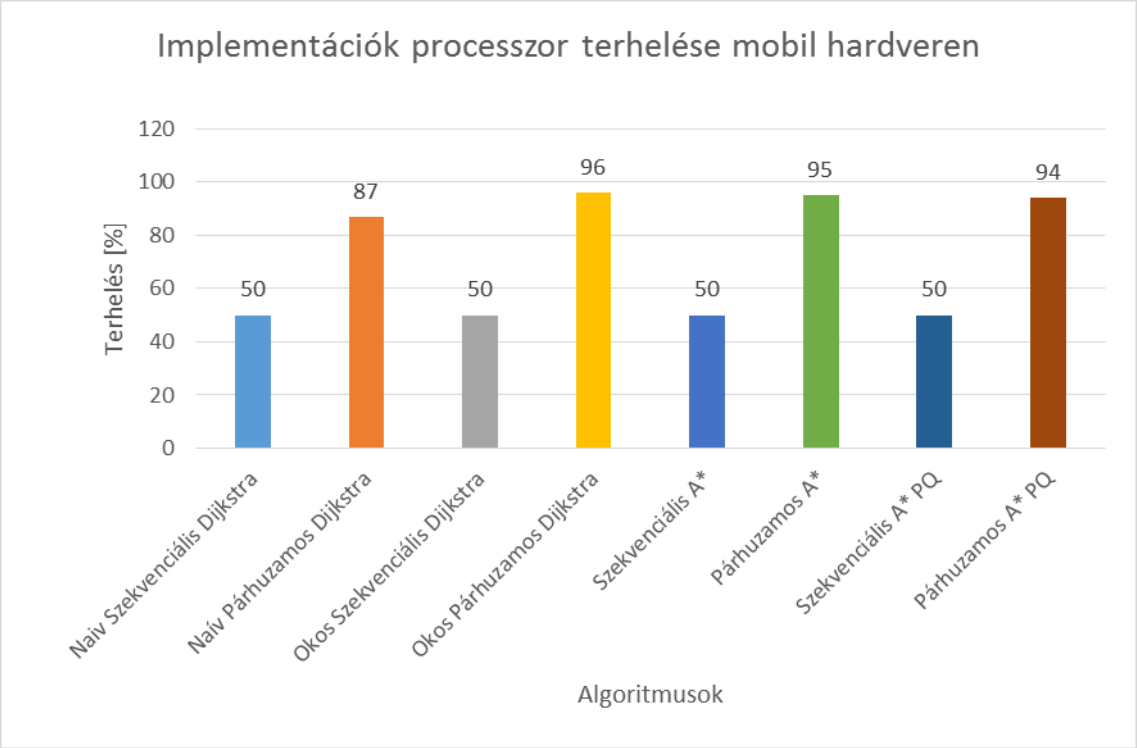
15. ábra Kliens és szerver összevetése

Ez lehet az a szint, ahol a szerver oldalon a több szálat még éppen csak ki tudja szolgálni a rendszer és az így keletkező adminisztrációs költség és a hozzá társuló hálózati kommunikáció hatására a mobil kliens meg tudja közelíteni a két szálú lokális futással.

6.4 Processzor használat

Ezzel a vizsgálattal arra szerettem volna fényt deríteni, hogy a párhuzamosítási módszerek milyen hatásfokkal működnek mobil platformon.

Mérési eredmények:



16. ábra Processzor kihasználtság mobil hardveren

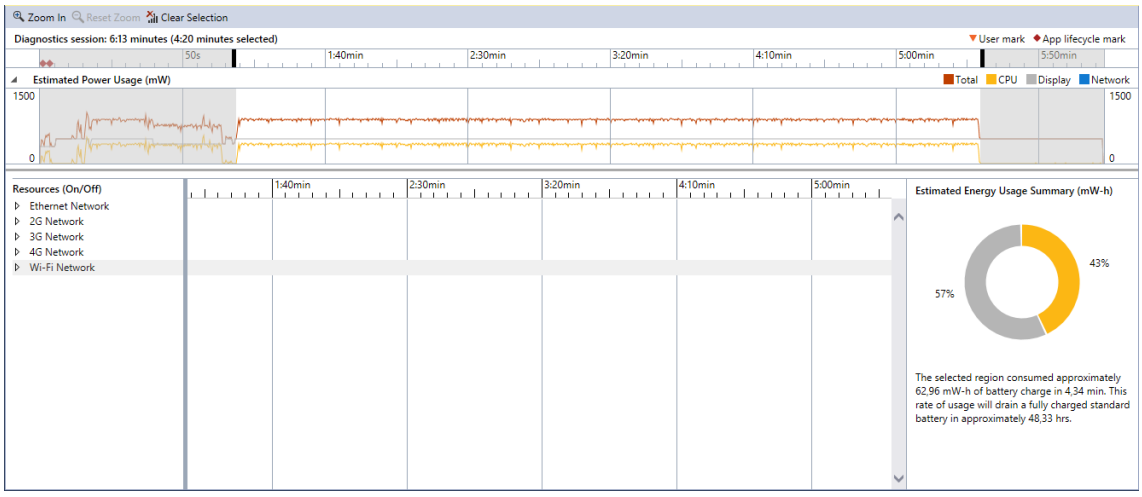
Értékelés:

Az értékek láttán elmondható, hogy legalábbis a vizsgált két szálas végrehajtásig a TPL jól skálázódik, hiszen kihasználja mindkét magot (≥50%).

6.5 Áramfogyasztás

A futási idő után a legfontosabb szempont a végrehajtás helyének választásakor, hogy mennyit is fogyaszt az alkalmazásunk. Szerver környezetben vehetjük úgy, hogy az áramfogyasztás költsége nulla, bele van kalkulálva az üzemeltetés költségébe. Mobil alkalmazásnál azonban nem szerencsés, ha a kis kapacitású akkumulátort indokolatlanul nagyon terheljük.

A fogyasztás méréseket a Visual Studio Performance and Diagnostic tool-jával végeztem, ami az alkalmazás készüléken való futása közben profilozza azt, adatokat gyűjtve az egyes komponensek (képernyő, processzor, hálózati egység) fogyasztásáról.

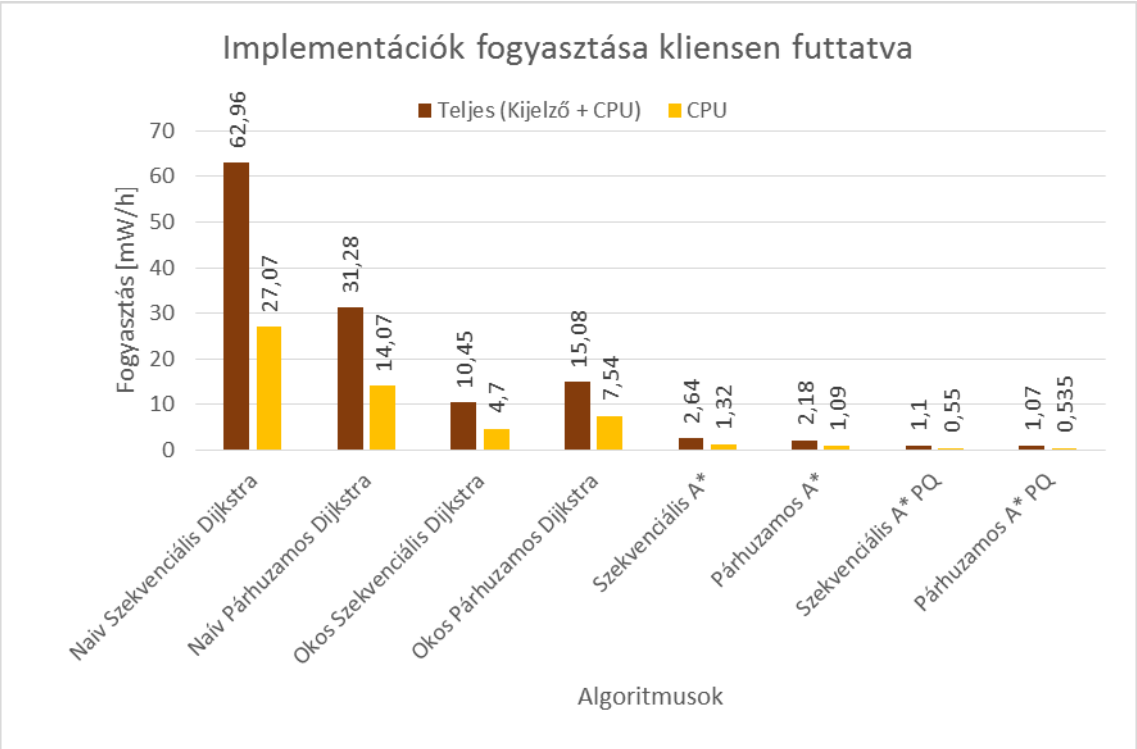


17. ábra Fogyasztás mérő felülete

A 10 ezer csúcsú gráf feldolgozásán rögzítettem a mérési adatokat.

6.5.1 Kliens lokális

Mérési eredmények:



18. ábra Fogyasztási adatok kliens lokális futtatásakor

Értékelés:

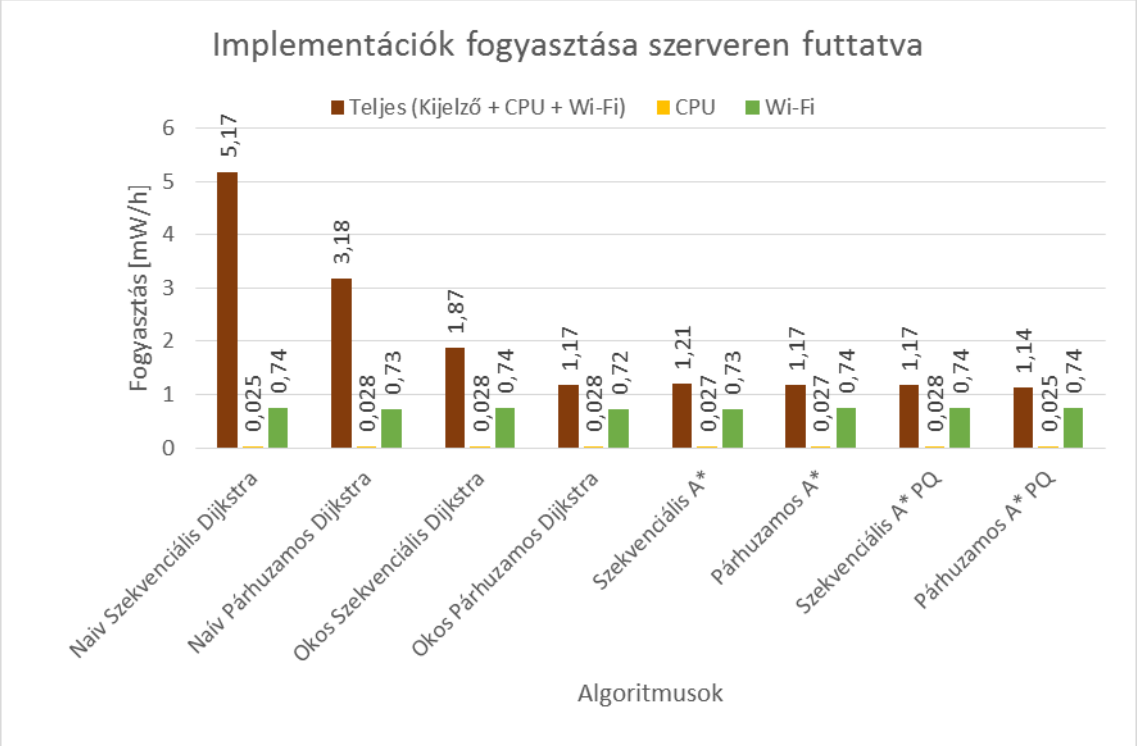
A kliensen való futtatás esetén a fogyasztás szinte egésze a kijelző és a központi egység munkájából adódik. Nagyon szembeötlő, hogy a teljes fogyasztás körülbelül felét a kijelző adja, így legalábbis ebben a kategóriában kijelenthető, hogy a kijelző

komolyabb fogyasztással rendelkezik, mint a processzor, mivel minden felhasználás esetén ennyi a fogyasztása.

A processzor fogyasztását vizsgálva megfigyelhető, hogy amikor 2 mag terhelődik a fogyasztása is megnő kisebb mértékben, azonban a lényeges láncszem az idővel való kapcsolat. Mivel két mag terhelése nem jár számottevő fogyasztás növekedéssel, a legjobban akkor járunk, ha a feladatot minél gyorsabban elvégezzük. Ez a futási idővel analóg tendenciát mutat. Levonható a következtetés, hogy a leggyorsabb végrehajtásra való törekvés nem csak a felhasználói komfort, hanem a fogyasztás csökkentése miatt is szükséges. Mindenképpen érdemes az algoritmusok optimalizálásába energiát fektetni, mert fogyasztás tekintetében a legjobb és a legrosszabb eredmény között közel 63-szoros a különbség.

6.5.2 Kliens – szerver

Mérési eredmények:



***19.* ábra Fogyasztási adatok Kliens - szerver esetén**

Értékelés:

Szerver oldali futtatásnál is elmondható, hogy a fő fogyasztó a képernyő, azonban ezt követően nem a processzor, hanem a hálózati modul a második legnagyobb

fogyasztó. A CPU fogyasztása szinte elhanyagolható, szerepe csupán a felhasználói felület kezelésében volt.

A mérések során ugyanazokat a teszt köröket használtam minden algoritmushoz, így lehetséges, hogy a Wi-Fi modul által elhasznált energia azonos, mivel minden esetben ugyanannyi adatot kellett kommunikálnia. Ebből látható, hogy kliens – szerver modell esetén közel sem mindegy a kommunikáció intenzitásának mértéke. Nagyobb mennyiségű adat esetén könnyedén megugorhat a fogyasztás, még akkor is, ha maga a feladat gyorsan lefutott.

A konkrét fogyasztási értékeket megvizsgálva elmondható, hogy lassabban, de ismét követi a lefutási idők alakulásának skáláját. Azért nem tud egészen ellaposodni, mint a kliens oldali kód esetében, mert a hálózati kommunikáció mindenképpen jelen van az alkalmazásban.

6.6 Hálózati kommunikáció

A hálózati kommunikáció elsősorban azért került a vizsgált kritériumok közé, mert még sokan rendelkeznek korlátos mobil internet hozzáféréssel és a szolgáltatók komoly árat kérnek a leforgalmazott adatmennyiségért cserébe. Ez elgondolkodtató tényező lehet kliens – szerver alkalmazás használata esetén.

A hálózati kommunikáció mérését az alkalmazásba építve valósítottam meg, kiolvasom az elküldendő http kérés tartalmának a méretét, majd mikor megjön a szervertől a válasz azon is elvégzem a kiolvasást és összegzem az értékeket, ezzel megkapva egy kérés-válasz pár méretét. Másik lehetőségem lett volna a teljes hálózati adatforgalom vizsgálata például a Wirehark nevű eszközzel.

Ahogy az előző fejezetben említettem, ugyanazokat a lekérdezéseket hajtottam végre minden esetben, így a hálózati forgalom nem tért el a vizsgált esetekben. Egy teszt kör után az átlagos üzenetváltás mérete 150 bájt körül alakult, ez az érték egy 10-15 hosszú úthoz tartozik.

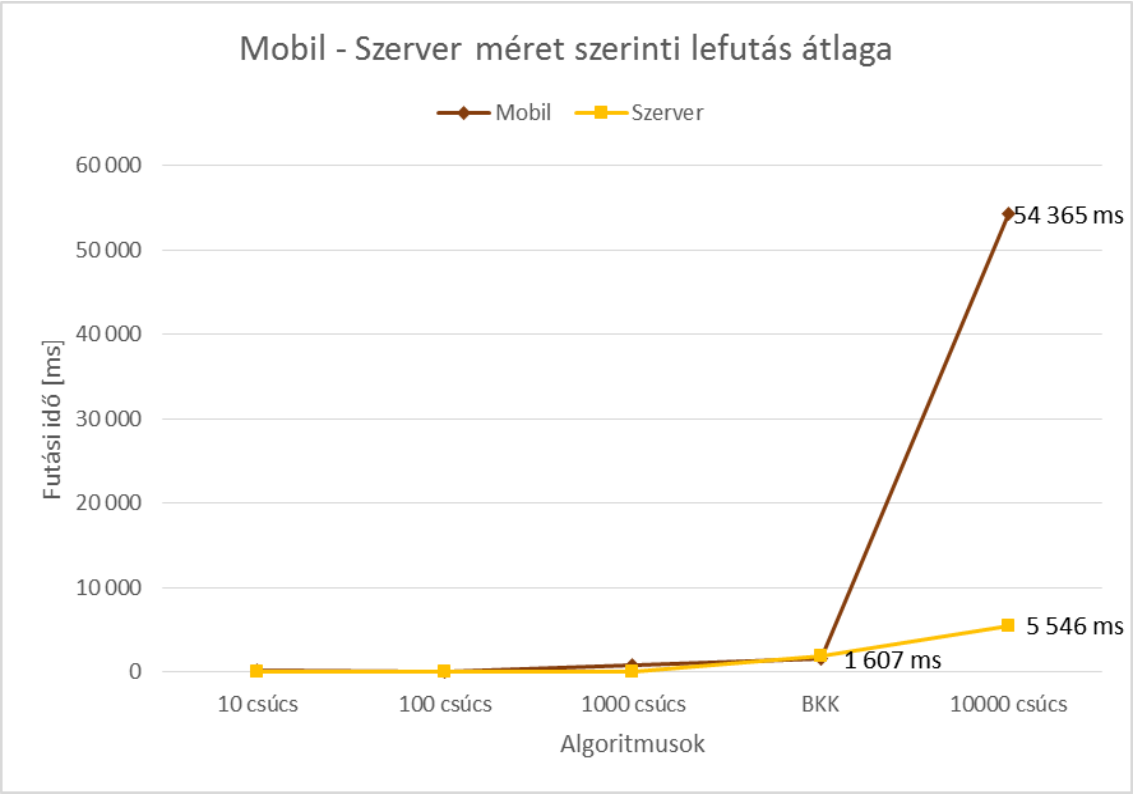
Elmondható, hogy a hálózati kommunikációnak mind a lefutási időre, mind a fogyasztásra hatása van. Minél több az adat küldés és fogadás annál lassabb a lefutási idő a kommunikációs overhead miatt és minél lassabb a feladat befejezése és minél több a forgalmazandó adat annál nagyobb a fogyasztás.

7 Konklúzió

A fentebb vázolt eredmények és következtetések után ideje, hogy ítéletet mondjak a kizárólag mobil klienst és kliens – szerver működést felhasználó alkalmazás felett. A három fő vizsgált terület a futási idő, fogyasztás és hálózati kommunikáció közül kiemelném a futási időt, mivel felhasználói szempontból ez a legfontosabb és nagymértékben befolyásolja a második helyen álló fogyasztás faktort is. A hálózati kommunikáció az utolsó helyre szorult, mivel lokális esetben egyáltalán nem kell vele számolni és egy-két kirívó esettől eltekintve a kliens – szerver működés során sincs nagy befolyásoló ereje, tolerálható mennyiségű adatforgalom keletkezik. Továbbá a szolgáltatók egyre gyakrabban csak lassítják az adatforgalmat, ha átlépte az előfizető a szerződésben foglalt mennyiséget és nem számlázzák ki neki a többlet forgalmat.

Kezdjük a legegyszerűbb esettel. Abban az esetben, ha a 100 ezer csúcsú gráf feldolgozásával megegyező vagy még komplexebb feladatot kell megvalósítani egyértelműen a szerver a használandó platform az okos telefon korlátos memóriája és teljesítménye miatt (lassú lefutás, magas fogyasztás), továbbá a mobil kliens létjogosultsága is megkérdőjelezhető, mert ez már egy igen hosszú lefutási időkkel rendelkező kategória és nem valószínű, hogy egy átlag felhasználó a telefonján hasonló műveleteket követne nyomon.

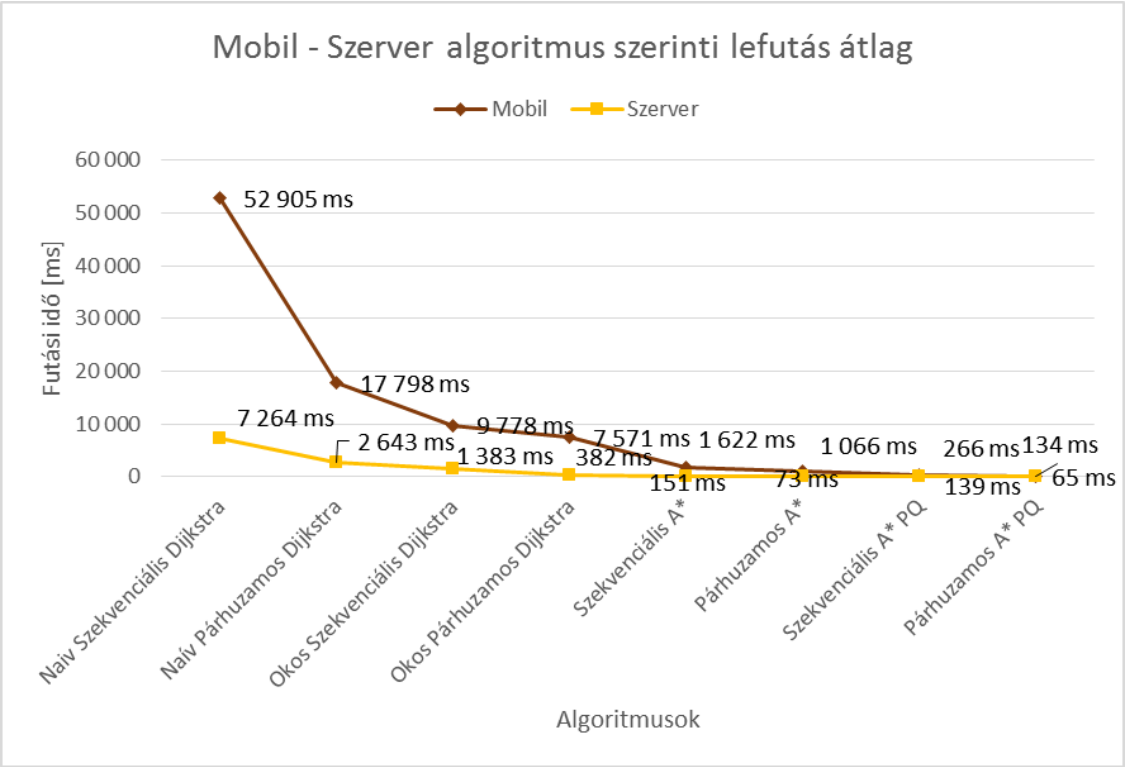
A 10 ezer csúcsú kategória az, ahol a két platform közül nem tudok egyértelműen jobbat választani. A jó döntés függ a problémától és a megoldására elérhető algoritmusoktól. A vizsgált esetben látható, hogy rosszabbul optimalizált algoritmusokkal a kliens oldali alkalmazás használhatatlan, mert nagyok a lefutási idők és nagy a fogyasztás. Ezzel szemben, ha megfelelően választunk algoritmust és implementációt (esetemben A* és változati) leszoríthatjuk mind az időbeli, mind a fogyasztásbeli költségeket a szerver oldali működés szintjére vagy akár alá és a hálózati kommunikáció is kikerül a képből. Ezzel egyértelmű favorittá válik, mivel a szerver üzemeltetési költségét is megspórolhatjuk.



20. ábra Mobil - Szerver méret szerinti lefutás átlaga

Ha az algoritmusok lefutási idejét átlagoljuk a különböző méretű gráfok szerint, akkor összességében az látható, hogy az éles eltérés a BKK gráfjánál található, innen nagy ugrás következik be a mobil oldali feldolgozásban, tehát ez lehet az a pont ahonnan felfelé már érdemes váltani a feldolgozási platformok között. Ebbe az eredménybe természetesen nagymértékben beleszól az olyan rosszul optimalizált algoritmusok gyenge eredménye, mint a Naiv Szekvenciális Dijkstra algoritmus. Ha ezeket elhagyjuk az összevetésből, akkor az előző bekezdés állítása továbbra is igaz marad.

Ez az állítás jól látható a következő ábrán ahol a futási idők átlagát az egyes algoritmusokra vetítve vettem fel. Látható, hogy az egyre jobban optimalizát algoritmusokkal a mobil kliens eredményei belesimulnak a szerver oldali feldolgozás eredményeibe, a használt gráfok méretétől függetlenül.



21. ábra Mobil - Szerver algoritmus szerinti lefutás átlaga

A 10 ezres kategória alatt, az eredmények tükrében a kliens oldali lokális feldolgozást választanám. A hálózati overhead miatt gyakorta előnyre is tesz szert a futási idők terén és végig képes tartani a még elfogadható reszponzivitási szintet. Továbbá a gyors futás és az internet használat hiánya miatt a fogyasztás sem sokkal emelkedik a fölé a szint fölé, amit az aktív képernyő egyébként is megkívánna. Természetesen további idő és energia nyerhető, ha itt is gondosan választjuk meg a felhasznált algoritmust.

Jól látható tehát, hogy való életbeli feladatok megoldására, mint például egy városban való navigálás (BKK gráf), már az alsókategóriás mobil eszközök is képesek, ráadásul teszik mindezt real-time körülmények között. Azokra a feladatok, ahol a mobil kliens már nem nyújt elegendő teljesítményt a pár másodpercen belüli válaszadásra, nagy valószínűséggel a szerver se tudja teljesíteni real-time a kérést, így a kiszervezés nem megoldás erre az esetre. Más eszközökhöz kell folyamodni, ilyen lehet például az ország szintű vagy országok közötti navigálás esetén a gráf hierarchiai szintekre bontása és az egyes szinteken a mobil eszköz újra alkalmas lesz a feladatra.

8 Továbblépési lehetőségek

A mobil készülékek egyre nagyobb térnyerésével és fejlődésével a felhasználók egyre több igényét tudják kielégíteni. Az eszközgyártók pedig egyre fejlettebb és erősebb hardverekkel szerelik csúcs készülékeiket így versenyezve a vásárlók kegyeiért, ezzel egy időben pedig az alsóbb kategóriákban is fejlődnek a hardverek.

A felső kategóriában a közelmúltban és az elkövetkezendő években egyre több, akár 8-10 magos processzorral szerelt hordozható eszköz is elérhetővé válik, amelyek mellé átlagosan már több, mint 2GB rendszer memóriát helyeznek el. Így a korábban általam meghatározott gazdaságossági határ könnyen még feljebb tolódhat. E mellett a grafikus vezérlők is fejlődnek és heterogén rendszert alkotva a CPU-val ezek az egységek is befoghatók általános számítási feladatokra.

További vizsgálatok egyik iránya lehet a felső kategóriás okos telefonok vagy tabletek teljesítményének feltérképezése, ebben a kategóriában a hagyományos számítógépek teljesítményét jobban megközelítő eszközökkel találkozhatunk.

Másik irányként a szolgáltatás felhő alapú rendszerbe migrálása jöhet számításba. Megvizsgálandó, hogy milyen előnyökkel jár, hogy nem vagyunk fix hardverhez kötve, hanem dinamikusan, az igényeknek megfelelően állíthatók be további virtuális gépek és építhetők ki terhelés elosztó fürtök, majd később ezek le is bonthatók, ahogy a rendszer terhelése változik. Figyelembe kell venni azonban bizonyos hátrányos tulajdonságokat is, mint például nem befolyásolhatjuk a gépeink hálózatban elfoglalt helyét, a szolgáltató felé nagy késleltetéssel is szembesülhetünk. Valamint a rendszerek közötti váltás koránt sem ígérkezik zökkenőmentesnek, ugyanis nincsenek kiépítve kellő integrációs segédeszközök.

Ebben az írásomban a statikus módszerrel, egy adott probléma körben mozogva igyekeztem meghatározni azt a határt, ahol a számításokat már érdemesebb kiszervezni a mobil készülékről. Ezzel megismerhettem a koncepció alapjait és képet kaptam az elérhető előnyökről és nehézségekről. A későbbiekben át szeretnék térni dinamikus döntéshozatalt alkalmazó rendszer fejlesztésére, amiben monitorozó algoritmusok futás időben döntenek a kiszervezés mértékéről és mikéntjéről, továbbá a mobil alkalmazás maga végzi a szükséges erőforrások foglalását.

9 Irodalomjegyzék

[1] Horváth Gábor: *Számítógép Architektúrák*, http://www.hit.bme.hu/~ghorvath/szgarch/book/szga.pdf (revision 09:33, 06 Szeptember 2015)

[2] Herédi Péter: *Párhuzamos útvonaltervezés gráf algoritmusokkal*, Budapesti Műszaki és Gazdaságtudományi Egyetem, BSc Szakdolgozat, 2015

[3] Microsoft: *Task Schedulers*, http://msdn.microsoft.com/en-us/library/dd997402%28v=vs.110%29.aspx (revision 18:42, 01 Szeptember 2015)

[4] Rónyai Lajos, Ivanyos Gábor, Szabó Réka: *Algoritmusok.* Typotex, Budapest, 2008 (115-120 o.)

[5] Zoltán Benedek, Cserkúti Péter: *Elosztott rendszerek jegyzet*, 2013

[6] Albet István: Mobilszoftver-platformok Windows diasorok

[7] Wiktor Jakubiuk, Keshav Puranmalka: *Parallelization of Dijkstra's Algorithm: Comparison of Various Priority Queues*, 2011. December. 14., http://www.jakubiuk.net/stuff/parallel_dijkstra.pdf

[8] Kevin Kelley, Tao B. Schardl: *Parallel Single-Source Shortest Paths*, http://courses.csail.mit.edu/6.884/spring10/projects/kelleyk-neboat-paper.pdf (revision 15:31, 09 November 2014)

[9] Jad Nohra, Alex J. Champandard: *The Secrets of Parallel Pathfinding on Modern Computer Hardware*, https://software.intel.com/sites/default/files/m/d/4/1/d/8/Secrets_of_Parallel_Pathfinding.pdf, (revision 17:21, 22 November 2014)

[10] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, Bharat Bhargava: *A Survey of Computation Offloading for Mobile Systems*, https://www.cs.purdue.edu/homes/bb/mobile-cloud-survey.pdf (revision 13:05, 10 Október 2015)

[11] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, Ellen Zegura: *COSMOS: Computation Offloading as a Service for Mobile Devices*, http://www.cc.gatech.edu/~khabak3/papers/COSMOS-MobiHoc'14.pdf revision 15:16, 10 Október 2015)

[12] Afnan Fahim, Abderrahmen Mtibaa, and Khaled A. Harras: *Making the Case For Computational Offloadingin Mobile Device Clouds*, http://reports-archive.adm.cs.cmu.edu/anon/qatar/CMU-CS-QTR-120.pdf (revision 17:08, 10 Október 2015)