Hatékony gyorsítótár használat legrövidebb útvonal kereséséhez helyzetlekérdező szolgáltatásoknál

Tanulmány: Effective Caching of Shortest Paths for Location-Based Services
Szerzők: Jeppe Rishede Thomsen, Man Lung Yiu, Christian S. Jensen

Gyimesi Gábor – GYGRAAI.ELTE
Fodor Krisztián – FOKQAAI.ELTE
Bodnár István – BOIQAAI.ELTE

Összefoglalás

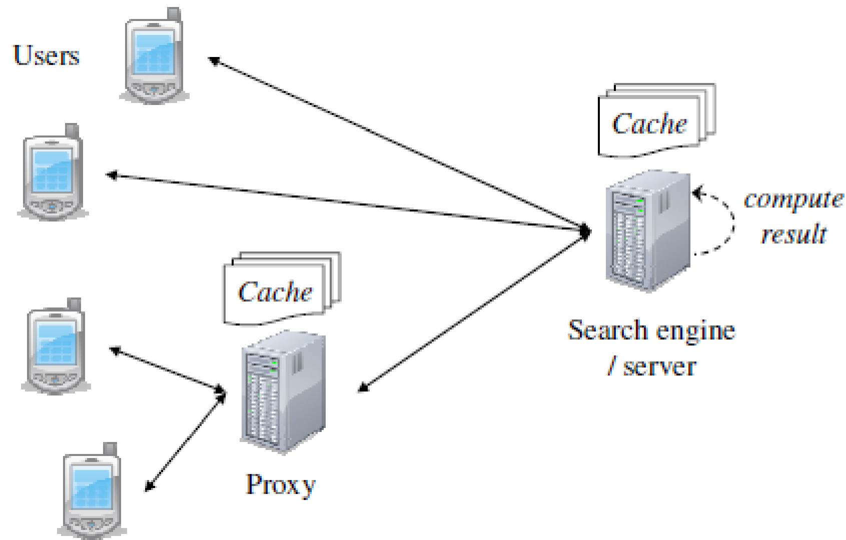
Mai életünk elengedhetetlen része a webes keresők használata. A keresőmotorok számítási idejének és a proxy szerverek közötti hálózati kommunikáció csökkentése érdekében a keresőmotorok gyorsítótárakat használnak. A webes keresések egy másik formája az online legrövidebb útvonalkeresés, amely az utóbbi időben nagyfokúan elterjedt a mobil eszközök megnövekedett használata és a GPS eszközök felgyorsult fejlődése miatt. Azonban a meglévő gyorsítótárazási technikák nem hatékonyak a legrövidebb útvonalak lekérdezésére. Ennek okai a webes keresések és az útvonalkeresések közötti különbségekből származnak. Ez alapján szükséges felismernünk azon tulajdonságokat, melyek elengedhetetlenek, hogy meghatározzunk egy hatékony gyorsítótárazási technikát a legrövidebb útvonalak meghatározásához. Ehhez kihasználjuk az optimális részútvonal tulajdonságot, amely lehetővé teszi a legrövidebb útvonal meghatározását bármely kezdő és végpontú útvonal lekérdezésekor. Felhasználunk továbbá statisztikai adatokat a lekérdezési naplókból, hogy megbecsüljük a gyorsítótár használatból származó előnyöket egy adott útvonal meghatározásánál, majd alkalmazunk egy mohó algoritmust, amely elhelyezi a gyorsítótárba a megfelelő útvonalakat. Emellett megtervezünk egy kompakt gyorsítótár adatszerkezetet, amely hatékonnyá teszi lekérdezéseinket.

Bevezetés

Manapság óriási mennyiségű internetes keresést futtatunk le. Ennek jellemző lefutása a következők szerint történik: 1. A felhasználó lefuttat egy lekérdezést pl.: „Párizs Eiffel-torony” a kereső szerveren, amely kiszámítja a releváns eredményeket és visszajuttatja azokat a felhasználóknak. A szerveren lévő gyorsítótár eltárolja a sokszor előforduló eredményeket, mellyel későbbi hasonló keresések gyorsan megválaszolhatók, ezzel csökkentve a számítási igényeket és javítva a válaszok megérkezésének sebességét.

A gyorsítótárat a számítások csökkentése érdekében jellemzően a keresőszerveren helyezzük el, ezzel meggyorsítva a választ abban az esetben, ha a válasz megtalálható a gyorsítótárban. Más megoldás a hálózati kommunikáció felgyorsítására, hogy a gyorsítótárat egy proxy szerveren helyezzük el, mely ugyanabban az alhálózatban található, mint a felhasználó. Így a keresés eredménye azonnal visszaadható, a keresőszerver elérése nélkül.

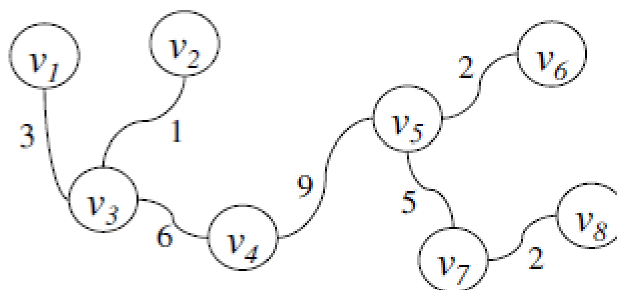
Az első képen láthatunk egy példát a legrövidebb út online keresésére. A megnövekedett mobil web használat és a térinformatikai technológiák fejlődése miatt, ez a webes lekérdezések egy népszerű formájává vált. Ez a típusú keresés lehetőséget ad a felhasználóknak múzeumokban, benzinkutakon, boltokban és éttermekben történő tájékozódáshoz.



1. Példa: Keresések lefutása

Ha összehasonlítjuk az offline navigációs rendszereket az online útvonal keresőkkel, mint például a Google Maps vagy a Mapquest, akkor arra a következtetésre juthatunk, hogy az utóbbiak sok előnnyel rendelkeznek a mobil felhasználók szemszögéből: (i) Ingyen szolgáltatásokat nyújtanak (ii) Nincs szükség telepítésükre a különböző mobil eszközökre (iii) Nincs szükség megvásárolni vagy telepíteni az újabb frissítéseket a térképadatok változása esetén.

Vegyünk egy úthálózatot, melyet a 2. példán láthatunk gráf formájában. Ezen a példán v_i egy útkereszteződés, (v_i,v_j) él pedig egy útszakasz ahol az él súlya a szakasz hosszát jelöli. Példaképp a legrövidebb út v_1 és v_7 között $<v_1,v_3,v_4,v_5,v_7>$ melynek távolsága 3+6+9+5=23. Itt is választhatunk, hogy gyorsítótárat a proxy szerveren helyezzük el, hogy csökkentsük a válaszidőt, vagy a szerveren, hogy csökkentsük a szerver oldali számítások idejét.



2. Példa: mintaútvonal gráfja

Most vessük össze ezt az útvonalkeresést az első példában látott esettel. Habár ez a séma alkalmazható mind a webes keresésekre mind a legrövidebb útvonal keresésére is, vannak meghatározó különbségek a két típusú keresés között, melyek a meglévő gyorsítótár használati technikákat alkalmatlanná teszik számunkra. A következőkben ezeket a különbségeket vizsgáljuk.

Pontos találatok vagy részútvonal találatok: Egy webes lekérdezésnél (pl.: „Párizs Eiffel-torony”) esetenként előfordul, hogy az eredmény megegyezik egy másik lekérdezés eredményeivel (pl.: „Párizs Louvre”). Ezzel szemben legrövidebb útvonalak keresésénél a pontos találatok helyett az eredmények tartalmazhatnak részútvonalakat, amelyeket felhasználhatunk más lekérdezések eredményeinek kiszámításához. Vegyük például a v_1 és v_7 közötti útvonalat ($<v_1,v_3,v_4,v_5,v_7>$) amely tartalmazza a legrövidebb útvonalat v_3 és v_5 vagy a v_4 és v_7 között is. Ez egy fontos tulajdonság melyet ki kell használnunk módszerünk megalkotásánál.

- **Gyorsítótár struktúrája**: A webes kereséseknél alkalmazható egy hash tábla az eredmények gyors elérése érdekében, gyorsan meghatározni, hogy az adott eredmény megtalálható-e a táblában. Azonban, egy ilyen hash tábla nem használható a mi esetünkben útvonalkeresésre ugyanis a részútvonalakat így nem tudjuk meghatározni. Szükségünk lesz egy új adatstruktúrára, amiben hatékonyan tudjuk szervezni a gyorsítótár tartalmát figyelembe véve a részútvonalakat. Ez esetben oda kell figyelnünk a részútvonalak átfedésére is, amely egy újabb kihívás az adatstruktúra tervezése során. Például ha vesszük a $<v_1,v_3,v_4,v_5,v_7>$ és a $<v_2,v_3,v_4,v_5,v_6>$ útvonalat máris találunk egy jelentős háromelemű $<v_3,v_4,v_5>$ részútvonalat, amely átfedés a két szakasz között. Ezt a tulajdonságot is ki fogjuk használni a gyorsítótár tervezése során.

- **Lekérdezés feldolgozásának költsége**: A szerver oldalon, ha a keresett útvonal nem található meg a cache-ben, meghívunk egy algoritmust az útvonal kiszámításának érdekében. Egyes eredmények kiszámítása többe kerül, mint másoké. Ahhoz, hogy optimalizáljuk a szerver oldali teljesítményt, szükségünk van egy modellre, hogy megbecsülhessük az egyes lekérdezések költségét. Azonban eddigi ismereteink alapján nem létezik ilyen modell határozatlan keresési algoritmusú útvonalkereséshez. Ahhoz, hogy a fenti kihívásokon felülkerekedjünk, a következőket tesszük:

- Előállítunk egy szisztematikus modellt az egyes útszakaszok gyorsítótár használatból származó előny meghatározására. Ezzel képesek leszünk egy pontos számot adni arról, hogy egy adott eredmény eltárolása, mennyiben előnyös a számunkra.

- Megtervezzük azon technikákat, melyekkel statisztikákat állíthatunk elő a lekérdezési naplókból, és megbecsülhetjük a legrövidebb út kiszámításainak költségeit.

- Megtervezünk egy algoritmust, mellyel meghatározzuk a gyorsítótárba elhelyezni érdemes útszakaszokat.

- Kifejlesztünk egy kompakt és hatékony gyorsítótár adatstruktúrát a legrövidebb utak eltárolására.

- Tanulmányozzuk a fentieket valódi adatokkal történő tesztekkel.

Kapcsolódó munkák

Webes keresési gyorsítótárak: A webes keresések során lekérjük a felső K releváns találatot (pl.: weboldalakat) amelyek legjobban megfelelnek a lekérdezésnek. K értéke a találatok száma (pl.: 10) amennyit megjelenítünk egy oldalon; a következő oldali eredmények már egy más lekérdezéshez tartoznak.

A webes kereséseknél használt gyorsítótárakat a keresőmotor teljesítményének növelése érdekében használják. Ha egy lekérdezés megválaszolható a gyorsítótár segítségével, a lekérdezés kiszámításának költsége megspórolható. Markatos tanulmányában [16] olvashatunk a gyorsítótárak kétféle megközelítéséről. Az első a **dinamikus gyorsítótárak**, amelyek célja, hogy eltároljuk a legutóbb futtatok keresések eredményeit. Ennek egy módszere a legutóbb-használt (LRU), mely során minden újabb lekérdezés, amelynek eredményei nem találhatóak a gyorsítótárban bekerülnek oda, lecserélve a régebben használt eredményeket. Ez a megközelítés mindig frissen tartja a gyorsítótárat, azonban elég költséges a folyamatos frissítés miatt.

Egy másik megközelítés a **statikus gyorsítótár** használat. Ez a megközelítés a legnépszerűbb találatokat használja fel. Ez esetben a lekérdezési naplók tanulmányozásának segítségével meghatározzuk a legtöbbször előforduló találatokat. Az [1-3,16-18] tanulmányok kimutatták, hogy a lekérdezések gyakorisága Zipfian-eloszlást követ. Habár a gyorsítótár tartalma nem a legfrissebb, mégis képes megválaszolni a lekérdezések nagy részét. A statikus gyorsítótárat egy adott időközönként (például naponta) érdemes frissíteni.

Szemantikus gyorsítótár használat: Egy szerver-kliens rendszerben a kommunikációs költségek csökkentése érdekében a gyorsítótárat a kliens oldalon helyezzük el. Ebben a megoldásban a gyorsítótár csak a kliens oldali keresési eredményekkel rendelkezik, más kliensekhez nem fér hozzá. Az ebbe a kategóriába tartozó technikák dinamikus cache-elést alkalmaznak. A szemantikus gyorsítótár használat egy kliens oldali módszer, amely során a gyorsítótárban található eredmények részintervallumait felhasználva gyorsítjuk fel a keresést. A szemantikus keresés során egy Q lekérdezésben megkapunk egy részeredményt a gyorsítótárból, majd ez alapján konstruálunk egy Q’ lekérdezést, mellyel az eredmények hiányzó részeit megkaphatjuk és ezt a lekérdezést továbbírjuk a szervernek.

Legrövidebb útvonal kiszámítása: A meglévő útvonalkereső indexeléseket három kategóriába sorolhatjuk, amelyek az előszámítások mennyiségében és gyorsítótár használatban különböznek.

Első típus az **informálatlan keresések** (pl.: Dijkstra algoritmus) amelyet legrövidebb útvonalkeresésekre használhatunk, azonban nagyon nagy költségekkel jár.

Második típus a **teljesen informált keresések**, például Distance-vektorok alkalmazása, amelyben előre kiszámítjuk minden csúcsból az elérhető csúcsok távolságát a gráfban. Habár ez hatékony lekérdezési sebességgel szolgál számunkra, az előszámítási munkák hatalmas költséggel ($O(|V|^3)$) és hatalmas tárolóhely-használattal járnak.

Harmadik típus a **részben előszámított indexek** használata. Számunkra ez lehet a leghasznosabb, hiszen ebben ötvözhetjük a felső két módszer előnyeit.

Alapfogalmak

1. Definíció (Gráf modell): Legyen G(V,E) egy gráf ahol V a csúcsok halmaza és E az élek halmaza. Minden $v_i \in V$ csúcs a modellben egy útkereszteződést jelöl. Minden $(v_i,v_j) \in E$ él egy útszakaszt jelöl, melynek éle (hossza) $W(v_i,v_j)$-vel jelölt.

2. Definíció (Legrövidebb útvonal): lekérdezés és eredmény. Egy legrövidebb útvonal lekérdezést, jelöljünk $Q_{s,t}$-vel, ami egy v_s kezdőpontból és egy v_t végpontból áll. A $Q_{s,t}$ eredményét, jelöljük $P_{s,t}$-vel, a v_s-ből v_t-be (G gráfon keresztül) vezető legkisebb összsúllyal (hosszal) rendelkező utat. $P_{s,t}$-t reprezentálhatjuk csúcsok listájával: $<v_{x0},v_{x1},v_{x2},\ldots,v_{xm}>$, ahol $v_{x0} = v_s$, $v_{xm} = v_t$ és az út hossza: $\Sigma_{i=0\ m-1}\ W(v_{xi},v_{x1+1})$.

A mi példáinkban irányítatlan gráfokkal foglalkozunk. Az alkalmazott technikák később alkalmazhatóak irányított gráfokra is. A második képen vett példánkban a legrövidebb út v_1 és v_7 között $P_{1,7} = <v_1,v_3,v_4,v_5,v_7>$ melynek hossza 3+6+9+5=23. A legrövidebb útvonal keresése tartalmazza az optimális részútvonalak keresésének eredményeit is (lásd 1. lemma): a legrövidebb út minden részútvonala is egy legrövidebb út. A 2. kép példája alapján: $P_{1,7} = <v_1,v_3,v_4,v_5,v_7>$ tartalmazza az alábbi legrövidebb utakat: $P_{1,3}$, $P_{1,4}$, $P_{1,5}$, $P_{1,7}$, $P_{3,4}$, $P_{3,5}$, $P_{3,7}$, $P_{4,5}$; $P_{4,7}$; $P_{5,7}$.

1. Lemma (Optimális részútvonal tulajdonság): A legrövidebb út $P_{a,b}$ tartalmaz egy legrövidebb $P_{s,t}$ útvonalat ha $v_s \in P_{a,b}$-nek és v_t is $\in P_{a,b}$-nek. Pontosabban, legyen $P_{a,b} = <v_{x0},v_{x1},v_{x2},\ldots,v_{xm}>$. $P_{s,t} = <v_{xi},v_{xi+1},\ldots,v_{xj}>$ ha $v_s=v_{xi}$ és $v_t = v_{xj}$ ahol i,j-re $0\leq i<j\leq m$.

Ezt a tulajdonságot fogjuk kihasználni hamarosan, amikor a legrövidebb útvonalak gyorsítótár használatát szervezzük meg.

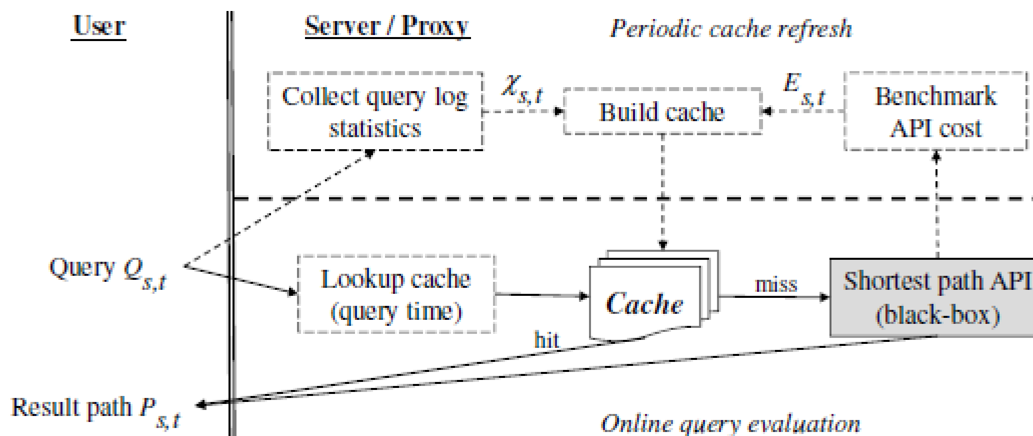
Célunk megfogalmazása során azt az architektúrát alkalmazzuk, amelyet az első képen láthattunk. Esetünkben a felhasználók kérdeznek le útvonalakat a szerverről mobil eszközeik segítségével. A gyorsítótárat, ahogy alább definiáljuk, elhelyezhetjük a szerveren vagy egy távoli gépen. Ezzel fogjuk csökkenteni mind a számítási költségeket, mind a kommunikációs költségeket a lekérdezések során.

3. Definíció (Gyorsítótár és kapacitás): Legyen a gyorsítótár kapacitásunk ß, a gyorsítótár pedig Ψ. A gyorsítótár olyan útvonal lekérdezési eredményeket tárolhat el melyre teljesül, hogy $|\Psi| \leq$ ß, ahol a gyorsítótár mérete $|\Psi|$ és a Ψ-ben található útvonalakon lévő csúcsok számát jelzi.

4. Definíció (Lekérdezési napló): A lekérdezési napló QL egy időponttal párosított, felhasználók által közelmúltban küldött lekérdezések gyűjteménye.

3. példában láthatjuk a szükséges komponenseket a gyorsítótár rendszerünkben: (i) a legrövidebb út API, (ii) a gyorsítótár, (iii) egy online modul a gyorsítótár lekérdezésére, és (iv) offline/időközönként meghívott modul a lekérdezési naplók begyűjtésére, mellyel megbecsülhetjük az API hívás költségeit és feltölthetjük a gyorsítótárat.

A szürkén jelölt legrövidebb út komponens a rendszeren kívül található, így azt nem módosíthatjuk az implementáció során. A szerver oldali tárolás esetén a legrövidebb útvonalat számító API egy tipikus útvonalkereső algoritmust tartalmaz (pl.: Dijkstra, A* algoritmus). A gyorsítótár proxy szerveren történő tárolása esetén a legrövidebb útvonalat számoló API egy lekérdezést küld a szerver felé. Mindkét esetben az API hívás drága kommunikációs/számítási költségekkel jár azonban különböző lekérdezések, különböző költségekkel járnak, így később becsléseink alapján megállapíthatjuk mely lekérdezési eredményeket érdemes eltárolnunk. Ezeket a költségeket a következőkben definiáljuk.



3. Példa: Gyorsítótár komponensei

5. Definíció (Lekérdezés végrehajtásának költsége): Jelöljük $E_{s,t}$-vel a legrövidebb út API hívásának költségét (pl.: válaszidő) amely feldolgozza a $Q_{s,t}$ lekérdezést.

Olyan gyorsítótárat alkalmazunk, amellyel csökkenthetjük az API hívásának költségét. Miután megkapjuk a lekérdezést (futási időben), a szerver/proxy ellenőrzi, hogy a gyorsítótár tartalmazza-e a lekérdezés eredményét. Ha megtalálható, az eredményt azonnal küldhetjük a felhasználónak. Ezzel megspóroljuk az API hívás költségét. Más esetben, azonban az eredményt az API hívás eredményeként kaphatjuk meg.

Azt vehetjük észre, hogy gyorsítótár eredményességének maximalizálása nem feltétlen csökkenti nagymértékben az összköltségünket. Szerver oldali tárolás esetén az API hívás (pl.: egy legrövidebb útvonalat számoló algoritmus) költsége nem állandó, mivel nagymértékben függ a kezdő- és végpont távolságától. Ahogy a tanulmányok is kimutatták korrelációt fedezhetünk fel a számítási költségek és a távolság között.

A fő célunk, hogy lecsökkentsük a legrövidebb utat számító API hívásának összköltségét. Ezt a problémát alább definiáljuk. Ehhez a továbbiakban definiálunk egy mértékegységet, mellyel jellemezhetjük a gyorsítótárba helyezés előnyösségét, statisztikai adatokat elemzünk ennek meghatározására és megadunk egy algoritmust a gyorsítótárból származó haszon maximalizálása érdekében.

Probléma: Statikus gyorsítótárból származó haszon maximalizálása: Egy megadott ß költség és QL lekérdezési napló mellett, állítsunk elő egy Ψ gyorsítótárat $\gamma(\Psi)$ maximális haszonnal, a ß megszorítás mellett, ahol Ψ tartalmazza azon $P_{s,t}$ eredményeket melynek $Q_{s,t}$ lekérdezései a QL lekérdezési naplóhoz tartoznak.

Másodlagos célunk: (i) kifejlesszünk egy kompakt gyorsítótár struktúrát, amely maximalizálja az elhelyezhető útvonalakat, és (ii) hatékony megoldást nyújt a benne lévő adatok elérésére.

Már meglévő gyorsítótár használati technikák

A saját gyorsítótár szerkezetünk megvalósításához definiálnunk kell a későbbiek során felhasznált már meglévő gyorsítótár használati technikákat, amit az alábbiakban megteszünk.

Dinamikus gyorsítótár-LRU: Egy tipikus gyorsítótár használati eljárás a webes keresésekre a Least-Recently-Used (LRU) azaz a legutóbb használt eredmények módszere. Amikor egy új lekérdezést küldünk a szervernek, annak eredményét eltároljuk a gyorsítótárban. Ha a gyorsítótár megtelik, akkor a legkevésbé használt eredményt eltávolítjuk és az újonnan kapott eredményt helyezzük el helyette.

Ismét felhasználjuk a 2. példán található gráfunkat, hogy bemutassuk az LRU típusú gyorsítótár használatot. Legyen a gyorsítótár ß kapacitása 10 (pl.: 10 csúcsot tud eltárolni). Az alábbi táblázat mutatja a lekérdezés és a gyorsítótár tartalmát minden T_i időpontban. Minden gyorsítótárban lévő útvonalhoz tartozik egy időpont, amikor utoljára használtuk azt. T_1 és T_2 időpontokban történt lekérdezések eredményének egyike sem található a gyorsítótárban, így azok eredménye bekerül. T_3-as időpontban a $Q_{2,7}$-es lekérdezés eredménye sem található meg a gyorsítótárban. Ezután mielőtt a $P_{2,7}$-es eredmény bekerülne a gyorsítótárba ki kell vennünk a legrégebben használt útvonalat, ami a $P_{3,6}$. T_4-es időpontban a $Q_{1,4}$-es lekérdezés eredménye kiolvasható a gyorsítótárból, mivel az ott található $P_{1,6}$ útvonal tartalmazza a v_1 és v_4-es csúcsokat.

Time	$Q_{s,t}$	$P_{s,t}$	Paths in LRU cache	event
T_1	$Q_{3,6}$	$\langle v_3, v_4, v_5, v_6\rangle$	$P_{3,6} : T_1$	miss
T_2	$Q_{1,6}$	$\langle v_1, v_3, v_4, v_5, v_6\rangle$	$P_{1,6} : T_2,\quad P_{3,6} : T_1$	miss
T_3	$Q_{2,7}$	$\langle v_2, v_3, v_4, v_5, v_7\rangle$	$P_{2,7} : T_3,\quad P_{1,6} : T_2$	miss
T_4	$Q_{1,4}$	$\langle v_1, v_3, v_4\rangle$	$P_{1,6} : T_4,\quad P_{2,7} : T_3$	hit
T_5	$Q_{4,8}$	$\langle v_4, v_5, v_7, v_8\rangle$	$P_{4,8} : T_5,\quad P_{1,6} : T_4$	miss
T_6	$Q_{2,5}$	$\langle v_2, v_3, v_4, v_5\rangle$	$P_{2,5} : T_6,\quad P_{4,8} : T_5$	miss
T_7	$Q_{3,6}$	$\langle v_3, v_4, v_5, v_6\rangle$	$P_{3,6} : T_7,\quad P_{2,5} : T_6$	miss
T_8	$Q_{3,6}$	$\langle v_3, v_4, v_5, v_6\rangle$	$P_{3,6} : T_8,\quad P_{2,5} : T_6$	hit

1. Táblázat: LRU lekérdezések

A mi esetünkben ez átalakításra szorul, mivel a LRU nem képes meghatározni a meglévő útvonalak hasznosságát. Például $P_{1,6}$ és $P_{2,7}$ útvonalak (amelyeket T_2 és T_3-as időpontban szereztünk) képesek megválaszolni az utolsó négy egymást követő lekérdezést: $Q_{1,4}$;$Q_{2,5}$;$Q_{3,6}$;$Q_{3,6}$. Ha ezeket megtartanánk a gyorsítótárban az négy egymást követő találattal szolgálna számunkra. Az LRU módszer eltávolítaná ezeket az eredményeket mielőtt hasznukat vehetnénk.

Másik hátránya az LRU-nak, hogy nem támogatja a részútvonalak közötti keresést. Miután megkaptuk a $Q_{s,t}$ lekérdezést, minden meglévő útvonalat le kell ellenőriznünk, hogy tartalmazza-e v_s és v_t csúcsokat. Ez nagymértékű futási idő növekedéssel jár, amely csak háttérbe szorítaná a gyorsítótár használatának előnyeit.

Statikus gyorsítótár – HQF: Egy tipikus statikus gyorsítótár használati módszer az internetes keresésekhez a Highest-Query-Frequency (HQF), azaz leggyakoribb lekérdezések felhasználásának módszere. Az offline fázisban a legtöbbet használt lekérdezéseket a QL lekérdezési naplóból választjuk ki és annak eredményeit helyezzük el a gyorsítótárba. Futási időben a gyorsítótár tartalma változatlan marad.

Vegyünk egy példát webes gyorsítótár használatra, ahol a $Q_{s,t}$ lekérdezés gyakorisága a $Q_{s,t}$-vel megegyező lekérdezések száma QL-ben. Például legyen a lekérdezési napló QL = {$Q_{3,6}$;$Q_{1,6}$;$Q_{2,7}$;$Q_{1,4}$;$Q_{4,8}$;$Q_{2,5}$;$Q_{3,6}$;$Q_{3,6}$}. Mivel $Q_{3,6}$ a legnagyobb gyakoriságú (3), HQF az ehhez tartozó eredményútvonalat választja ki, amely a $P_{3,6}$. A $P_{1,6}$-ost nem választja ki, mivel $Q_{1,6}$-nak alacsony a gyakorisága (1). Azonban $P_{1,6}$ sokkal ígéretesebb mint $P_{3,6}$, mivel $P_{1,6}$ több lekérdezés megválaszolására felhasználható mint $P_{3,6}$. Ezzel meg is találtuk az egyik problémát a HQF módszerben mivel a lekérdezések gyakorisága nem hordozza magában azokat a tulajdonságokat, melyek számunkra fontosak lennének – a legrövidebb útvonalak között átfedés merülhet fel így egy lekérdezés eredménye több más lekérdezésnek is eredményt szolgáltathat.

LRU és HQF közös korlátai: Se LRU, se HQF nem veszi figyelembe az útvonalak kiszámításának költségeit. Vegyük a szerver oldali tárolás példáját. Intuitívan megállapítható, hogy sokkal költségesebb egy nagyobb távolságú lekérdezést megválaszolni, mint egy rövidebbet. Egy nehezen kiszámítható útvonal eltárolása pedig nagyobb költségmegtakarítással szolgálhat a későbbiekben. Egy informált útvonalválasztás esetén ezeket a költségeket is számba kell vennünk.

Másfelől az eddigi megközelítések nem vették figyelembe a gyorsítótár tárhelyének kapacitását a legrövidebb útvonalak tárolásakor. Például a 1. táblázatban, az útvonalak a gyorsítótárban átfedik egymást így helyet pazarolnak. Olyan kompakt gyorsítótárat kell terveznünk, amely kihasználja az átfedéseket és nem tartalmaz duplikált csúcsokat.

Eredmények

Hasznossági modell

Először megvizsgáljuk a gyorsítótárban elhelyezett útvonalak hasznosságát és utána a gyorsítótár hasznosságát. Vegyünk egy Ψ gyorsítótárat, amely csak a $P_{a,b}$ útvonalat tartalmazza. Emlékezzünk vissza, hogy a 3. példában amikor a $Q_{s,t}$ lekérdezés megválaszolható volt egy gyorsítótárban található $P_{a,b}$ útvonallal, akkor megspóroltuk az útvonalat kiszámító API meghívásának költségét. Ahhoz, hogy meghatározzuk a hasznosságát $P_{a,b}$-nek két kérdést vetünk fel:

1. Mely $Q_{s,t}$ lekérdezések válaszolhatók meg $P_{a,b}$-vel?
2. $Q_{s,t}$ lekérdezéssel mennyi költséget takarítunk meg?

Az első kérdés megválaszolható az 1. Lemma segítségével. A $P_{a,b}$ útvonal tartalmazza a $P_{s,t}$ útvonalat, ha mind v_s és v_t csúcs megtalálható $P_{a,b}$-ben. Tehát definiálhatjuk a megválaszolható lekérdezések halmazát $P_{a,b}$-vel a következőképpen:

$U(P_{a,b}) = \{ P_{s,t} \mid s \in P_{a,b} \wedge t \in P_{a,b} \wedge s != t\}$

Ha vesszük a 2. Példát mintagráfként, akkor a megválaszolható lekérdezések halmaza $P_{1,6}$-tal a következő halmaz: U(P1,6) = {P1,3; P1,4; $P_{1,5}$; $P_{1,6}$, $P_{3,4}$; $P_{3,5}$; $P_{3,6}$,$P_{4,5}$; $P_{4,6}$; $P_{5,6}$}. A 2. Táblázatban láthatjuk a többi útvonal segítségével megválaszolható lekérdezések halmazát.

$P_{a,b}$	$\mathfrak{U}(P_{a,b})$
$P_{1,4}$	$P_{1,3}, P_{1,4}, P_{3,4}$
$P_{1,6}$	$P_{1,3}, P_{1,4}, P_{1,5}, P_{1,6}, P_{3,4}, P_{3,5}, P_{3,6}, P_{4,5}, P_{4,6}, P_{5,6}$
$P_{2,5}$	$P_{2,3}, P_{2,4}, P_{2,5}, P_{3,4}, P_{3,5}, P_{4,5}$
$P_{2,7}$	$P_{2,3}, P_{2,4}, P_{2,5}, P_{2,7}, P_{3,4}, P_{3,5}, P_{3,7}, P_{4,5}, P_{4,7}, P_{5,7}$
$P_{3,6}$	$P_{3,4}, P_{3,5}, P_{3,6}, P_{4,5}, P_{4,6}, P_{5,6}$
$P_{4,8}$	$P_{4,5}, P_{4,7}, P_{4,8}, P_{5,7}, P_{5,8}, P_{7,8}$

$\mathfrak{U}(\Psi)$, when $\Psi = \{P_{1,6}, P_{3,6}\}$
$P_{1,3}, P_{1,4}, P_{1,5}, P_{1,6}, P_{3,4}, P_{3,5}, P_{3,6}, P_{4,5}, P_{4,6}, P_{5,6}$

2. Táblázat: U($P_{a,b}$) és U(Ψ)

A második kérdésre válaszolva, az elvárt $Q_{s,t}$ lekérdezés megspórolt költsége függ (i) a lekérdezés $X_{s,t}$ gyakoriságától és (ii) a legrövidebb útvonalat kiszámító API hívásának $E_{s,t}$ költségétől. Ha a $P_{a,b}$ megválaszolja a $Q_{s,t}$ lekérdezést megtakaríthatjuk az $E_{s,t}$ költséget összesen $X_{s,t}$-szer, így összesen $X_{s,t}$*$E_{s,t}$ költséget takaríthatunk meg.

A két kérdés válaszait összegezve definiálhatjuk a $P_{a,b}$ útvonal hasznosságát a következőképpen:

$$\gamma(P_{a,b}) = \sum_{P_{s,t} \in U(P_{a,b})} X_{s,t} * E_{s,t}$$

A $\gamma(P_{a,b})$ útvonal hasznosság a következő kérdésre ad választ: „Ha a $P_{a,b}$ útvonal megtalálható a gyorsítótárban, mennyi költséget takarítunk meg összesen?”

Tegyük fel, hogy adottak $X_{s,t}$ és $E_{s,t}$ értékek minden (v_s,v_t) párra, ahogy az 4. Példában láthatjuk.

$\chi_{s,t}$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	/	0	0	1	0	1	0	0
v_2	0	/	0	0	1	0	1	0
v_3	0	0	/	0	0	3	0	0
v_4	1	0	0	/	0	0	0	1
v_5	0	1	0	0	/	0	0	0
v_6	1	0	3	0	0	/	0	0
v_7	0	1	0	0	0	0	/	0
v_8	0	0	0	1	0	0	0	/

(a) $\chi_{s,t}$ values

$E_{s,t}$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
v_1	/	2	1	2	3	4	4	5
v_2	2	/	1	2	3	4	4	5
v_3	1	1	/	1	2	3	3	4
v_4	2	2	1	/	1	2	2	3
v_5	3	3	2	1	/	1	1	2
v_6	4	4	3	2	1	/	2	3
v_7	4	4	3	2	1	2	/	1
v_8	5	5	4	3	2	3	1	/

(b) $E_{s,t}$ values

4. Példa: Példa $X_{s,t}$ és $E_{s,t}$ értékekre

Ezután kiterjesztjük számításainkat általános esetre – egy gyorsítótárra amely több útvonalat tartalmaz. Vegyük észre, hogy ha egy lekérdezés megválaszolható a Ψ gyorsítótár segítségével, akkor megválaszolható bármely $P_{a,b}$-vel amely Ψ-ben található. Tehát definiáljuk a lekérdezések Ψ által megválaszolható halmazát az $U(P_{a,b})$ uniójaként és eszerint definiáljuk a Ψ hasznosságát is.

$$U(\Psi) = \bigcup_{P_{a,b} \in \Psi} U(P_{a,b})$$

$$\gamma(\Psi) = \sum_{P_{s,t} \in U(\Psi)} X_{s,t} * E_{s,t}$$

A $\gamma(\Psi)$ gyorsítótár hasznossági mutató megválaszolja a következő kérdést: „A Ψ gyorsítótár használatával összesen mennyi költséget spórolhatunk?”

Az egységenkénti hasznosság: A hasznossági modell nem veszi számba a $|P_{a,b}|$ méretét. Tegyük fel, hogy van két útvonalunk $P_{a,b}$ és $P_{a',b'}$ amelynek ugyanakkora a hasznossága (pl.: $\gamma(P_{a,b}) = \gamma(P_{a',b'})$) és ahol $P_{a',b'}$ rövidebb útvonal mint $P_{a,b}$. Intuitívan megállapíthatjuk, hogy a $P_{a',b'}$ útvonalat választanánk $P_{a,b}$ helyett mivel $P_{a',b'}$ kevesebb helyet foglal, így más útvonalakat is eltárolhatunk a gyorsítótárban. Ezért bevezetjük a haszon-per-méret mutatót a $P_{a,b}$ útvonalra:

$$\bar{\gamma}(P_{a,b}) = \frac{\gamma(P_{a,b})}{|P_{a,b}|}$$

Későbbiekben ezt a mértéket fogjuk alkalmazni. Ahhoz, hogy maximalizáljuk a gyorsítótár hasznosságát, meg kell határoznunk a $X_{s,t}$ és $E_{s,t}$ értékét, ahogy az előbbiekben összefoglaltuk. Vegyük először $X_{s,t}$ meghatározását.

A $Q_{s,t}$ lekérdezés $X_{s,t}$ gyakorisága fontos szerepet játszik a hasznossági modellben. A [7] tudományos tanulmány alapján, a felhasználók mobilitási mintája aszimmetrikus mintát követ. Például a népszerű régiók (pl.: bevásárlóközpontok, panellakások) általában magas $X_{s,t}$-vel rendelkeznek, míg kevésbé népszerű régiók (pl.: vidéki régiók) valószínűleg alacsony $X_{s,t}$-vel rendelkeznek.

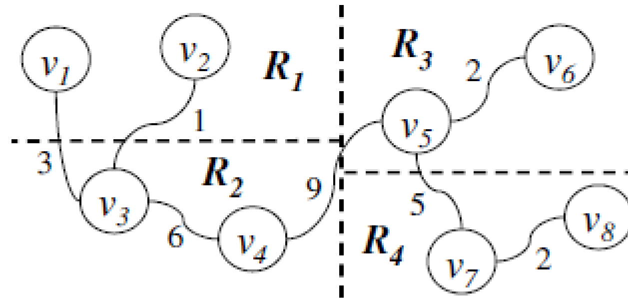
Szükségünk lesz egy automatizálható technikára, hogy meghatározzuk az $X_{s,t}$ értékeket. A mi gyorsítótár használati módszerünkkel a szerver/proxy megadott időközönként összegyűjti a QL lekérdezési napló tartalmát és kiszámítja a $X_{s,t}$ értékeket. A [2] tanulmány alapján a lekérdezési gyakoriság stabil marad egy hónapra, így ezt az időintervallumot egy hónapra állítjuk be. Először kipróbálunk egy egyszerű módszert $X_{s,t}$ megállapítására.

Csúcspár gyakoriság számlálása: Ezzel a módszerrel először előállítjuk a csúcspár gyakoriság táblázatot $|V|x|V|$ értékekkel, ahogy a 4a példában láthatjuk. Az s-edik sor t-edik oszlopa reprezentálja az $X_{s,t}$ értékét. A tárolási költsége a táblázatnak $O(|V|^2)$, a lekérdezési napló méretétől függetlenül.

Kezdetben minden érték a táblázatban 0. Ezután megvizsgálunk minden $Q_{s,t}$ lekérdezést a QL lekérdezési naplóból és növeljük a hozzájuk tartozó $X_{s,t}$ (és $X_{t,s}$) értékeket.

Régiópár gyakoriság számlálás: A X csúcspár gyakorisági táblázat $O(|V|^2)$ helyet foglal, amely nem fér be a memóriába egy teljes úthálózat esetén (pl.: $|V|$ = 100,000). Hogy megoldjuk ezt a problémát, (i) particionáljuk a gráfot L régióra (ahol L rendszerparaméter), és (ii) alkalmazunk egy kompakt táblázatot csak a régiópárok közötti lekérdezések tárolására.

Első lépésként felhasználunk egy létező gráf particionálási technikát (pl.: kD-fa particionálás, spektrális particionálás). A kD-fa particionálás alkalmazható az úthálózatok nagy részére, amelyek rendelkeznek csúcskoordinátákkal. Más gráfoknál, alkalmazhatjuk a spektrális particionálást, amelynek nincs szüksége csúcskoordinátákra. A 5a példában kD-fát alkalmazunk a csúcskoordináták segítségével, hogy L = 4 régióra bontsuk a gráfot: R1, R2, R3, R4.



(a) a graph partitioned into 4 regions

R_1 : $\{v_1, v_2\}$
R_2 : $\{v_3, v_4\}$
R_3 : $\{v_5, v_6\}$
R_4 : $\{v_7, v_8\}$

(b) node sets of regions

χ_{R_i,R_j}	R_1	R_2	R_3	R_4
R_1	0	1	2	1
R_2	1	0	3	1
R_3	2	3	0	0
R_4	1	1	0	0

(c) region-pair frequency table $\widehat{\chi}$

5. Példa: Régiópár gyakoriság számlálás

Második lépésként elkészítünk egy régiópár gyakorisági táblázatot LxL értékekkel, ahogy a 5c példában láthatjuk. Az R_i-edik sor R_j-edik oszlopában lévő érték az R_iR_j értéket reprezentálja. Ebben a táblázatban a helyfoglalás csak $O(L^2)$-es. Kezdetben minden értéket a táblázatban 0-ra állítjuk. Minden $Q_{s,t}$ lekérdezésre a QL lekérdezési naplóból megkeressük a régiót (pl.: R_i-t) amely tartalmazza a v_t csúcsot. Ezután megnöveljük a R_jR_i és R_iR_j értékét.

Utolsó lépésként meghatározzuk a $X_{s,t}$ kiszámításának módját a regiópár gyakorisági táblázatból. Vegyük észre, hogy R_i,R_j-t minden (v_s,v_t) párra megnöveljük, amelyre teljesül, hogy R_i tartalmazza v_s-t és R_j tartalmazza v_t-t. Így megkapjuk, hogy $\hat{X}R_i, R_j = \sum_{vs \in Ri} \sum_{vt \in Rj} X_{s,t}$. Ebből megkaphatjuk a régiókra, hogy $R_i,R_j = |R_i| * |R_j| * X_{s,t}$, ahol $|R_i|$ és $|R_j|$ az R_i és R_j-ben található csúcsok számát jelölik. Ez alapján a következőképpen számoljuk:

$$X_{s,t} = \frac{\hat{X}R_i, R_j}{|R_i| \cdot |R_j|}$$

$X_{s,t}$ értékét csak akkor számoljuk ki, ha szükség van rá. Nincs szükség felesleges helyfoglalásra $X_{s,t}$ előrefoglalása során.

Következőleg feladatunk definiálni az $E_{s,t}$ érték kiszámításának menetét a legrövidebb útvonalakat kiszámító API-hoz. A mi modellünkben akkor hívjuk meg a legrövidebb útvonalat

kiszámító API-t, ha nem található meg az eredmény a gyorsítótárban. A következőkben meghatározzuk, hogy állapíthatjuk meg az $E_{s,t}$ számítási költséget a $Q_{s,t}$ lekérdezéshez.

A proxy használat esetén az API egy lekérdezést küld a szervernek. A költség nagy része a kommunikációs költségből áll, amely ugyanannyi minden lekérdezésnél. Tehát a proxy esetében a következővel definiálhatjuk a költséget:

$E_{s,t}(\text{proxy}) = 1$

A szerver oldali számolás esetét jobban meg kell vizsgálnunk. Jelöljük az útvonalat kiszámító algoritmust ALG-gal $Q_{s,t}$ lekérdezéshez annak költségét pedig $E_{s,t}(\text{ALG})$-gal.

Kifejlesztünk egy általános technikát $E_{s,t}(\text{ALG})$ kiszámítására: ezt a technikát bármely ALG gráfalgoritmusra alkalmazhatjuk. Egy brute-force módszer lenne $E_{s,t}(\text{ALG})$ előre kiszámítása minden v_s-ből v_t csúcsba menő útvonal kiszámítására. Ezeket pedig eltárolhatnánk egy táblázatban, amit az 4b példában is láthatjuk. Azonban ez a megközelítés nagyon is költséges, mivel ehhez le kell futtatnunk az ALG algoritmust $|V|^2$ alkalommal.

A mi módszerünkkel csak kisebb előszámolásokat végzünk. Intuitívan megállapíthatjuk, hogy az $E_{s,t}$ költség erős korrelációban áll a $P_{s,t}$ útvonal hosszával. A rövid útvonalak kiszámítása kis $E_{s,t}$-vel míg a hosszú útvonalak kiszámítása nagy $E_{s,t}$-vel jár. A mi ötletünk, hogy osztályozzuk a lekérdezéseket távolságuk szerint és megbecsüljük a költségeket a kategória szerint.

Becslési struktúra: Becsléshez felépítünk két adatstruktúrát: (i) egy távolság becslőt és (ii) egy költség hisztogramot. A távolság becslő egyszerűen földrajzi koordináták alapján megbecsüli a távolságot. A költségi hisztogramot használjuk a lekérdezések átlagos költségének feljegyzésére a távolságokat figyelembe véve, ahogy a 7b példán láthatjuk. Általánosan a hisztogram H távolsági kategóriát tartalmaz. Lefuttatjuk az ALG algoritmust S véletlenszerű lekérdezéshez, hogy megállapítsuk azok költségét, majd frissítjük ezek alapján a hisztogramot. A hisztogram tárolási költsége Ơ(H) felépítésének költsége pedig S*Ơ(ALG). Ezek átlagos értéke: H=10 és S=100 egy átlagos úthálózaton.

A becslés menete: A fenti struktúrák segítségével, az $E_{s,t}$ értékét megbecsülhetjük 2 lépésben. Először használjuk a távolság becslést a [19]-es referencia alapján $P_{s,t}$ becslésére: $\min_{i=1..|U|} d(u_j,v_s) + d(u_j,v_t)$. Ezután megnézzük a költség hisztogramot és visszaadjuk a megfelelő költséget $E_{s,t}$ értékeként.

Gyorsítótár előállításának algoritmusa

Mint más statikus gyorsítótár használati módszereknél, itt is kihasználjuk a QL lekérdezési naplót, hogy megtaláljuk a megfelelő eredményeket, amiket elhelyezhetünk a Ψ gyorsítótárba. Az algoritmus lényege, hogy megállapítsuk azokat az útvonalakat, amelyek maximalizálják a gyorsítótár $\gamma(\Psi)$ hasznosságát ß korlát mellett.

A [17]-es referencia alapján mi is egy mohó algoritmust fogunk alkalmazni ehhez. Könnyen megoldható lenne, hogy a mohó megközelítéssel (i) kiszámoljuk a $(P_{a,b})$ méret-szerinti-haszont minden $P_{a,b}$-re utána (ii) iteratív módon elhelyezzük a legmagasabb $(P_{a,b})$ értékű elemeket a gyorsítótárba. Sajnos ez a megközelítés nem a legmegfelelőbb, mivel ez a megközelítés nem veszi figyelembe a már gyorsítótárban lévő elemeket.

A következő problémát megoldva egy olyan módszert alkalmazunk, amely előtérbe helyezi azt a $P_{a,b}$ útvonalat, amely képes olyan lekérdezéseket megválaszolni, amiket más gyorsítótárban lévő útvonalak nem tudnak.

6. Definíció: ($P_{a,b}$ Útvonal növekedési méret alapú haszna):

Legyen $P_{a,b}$ legrövidebb útvonal növekedési méret alapú haszna $\Delta(P_{a,b},\Psi)$ figyelembe véve Ψ gyorsítótárat. Definiáljuk a $P_{a,b}$-t Ψ gyorsítótárba elhelyezéséből származó hozzáadott hasznot $P_{a,b}$ mérete szerint a következőképpen:

$$\Delta\bar{\gamma}(P_{a,b},\Psi) = \sum_{P_{s,t}\in U(P_{a,b})-U(\Psi)} \frac{X_{s,t}\cdot E_{s,t}}{|P_{a,b}|}$$

Ez alapján előállítunk egy átgondolt mohó algoritmust. Kezdetben a Ψ üres. Minden körben az algoritmus kiszámítja a növekedési hasznot minden $P_{a,b}$ útvonalra a Ψ gyorsítótárban. Ezután az algoritmus kiválasztja a legnagyobb Δ értéket és elhelyezi azt Ψ-ben. Ezt addig folytatjuk amíg Ψ meg nem telik (például elérjük a ß korlátot). Erre láthatunk példát az 5-ös táblázatban.

A gyorsítótárat előállító algoritmus és annak időkomplexitása: Az algoritmus pszeudo-kódja mutatja a mohó algoritmust. Ennek bemenete egy G(V,E) gráf, a gyorsítótár korlátja ß, a lekérdezési napló pedig QL. A ß korlát jelöli a gyorsítótár által befogadható csúcsok számát. A X lekérdezési gyakoriság és az E lekérdezési költség a növekedési haszon kiszámításához szükséges.

Az algoritmus első 5 sora az inicializációs fázis. A Ψ gyorsítótár kezdetben üres. A H kupacot használjuk arra, hogy rendezzük az eredményeket csökkenő sorrendben Δ értékeik alapján. Minden $Q_{a,b}$ lekérdezésre a QL lekérdezési naplóból lekérjük a $P_{a,b}$ értéket, kiszámítjuk annak Δ értékét és elhelyezzük $P_{a,b}$-t H-ban.

Az algoritmus optimalizációt is végez, hogy csökkentse a növekedési haszon kiszámításának költségét minden körben (pl.: a 6-13. Sorban lévő ciklusban). Először kiválasztjuk azon $P_{a',b'}$ útvonalat H-ból, amelynek a legnagyobb Δ értéke. (7.sor) és kiszámoljuk az aktuális Δ értékét. A 2. Lemma alapján a $P_{a,b}$ útvonal Δ értéke H-ban, amelyet az előző körben számoltunk ki, az felső korlátja az aktuális körben kiszámított Δ értéknek. Ha $\Delta(P_{a',b'},\Psi)$ érték a H-ban lévő legelső érték felett van, akkor nyugodtan mondhatjuk, hogy $P_{a',b'}$ minden H-ban lévő értéknél jobb, anélkül hogy ki kellene számolnunk a pontos Δ értékeket. Ezután elhelyezzük $P_{a',b'}$ útvonalat Ψ gyorsítótárba ha van elég ß-|Ψ| hely még benne. Ha a $\Delta(P_{a',b'},\Psi)$ érték kisebb mint a H legfelső eleme, $P_{a',b'}$ értéket visszarakjuk H-ba. Amikor H üres lesz a ciklus terminál, és visszaadjuk a gyorsítótárat.

2. Lemma (Δ értéke folyamatosan csökken az i. körben): Legyen Ψ_i a gyorsítótár állapota mielőtt lefuttatjuk az i. kört. Ekkor teljesül, hogy: $\Delta(P_{a,b},\Psi_i) \geq \Delta(P_{a,b},\Psi_{i+1})$.

Algorithm 1 **Revised-Greedy**(Graph $G(V, E)$, Cache budget $\mathcal{B}$, Query log $\mathcal{QL}$, Frequency χ, Expense E)

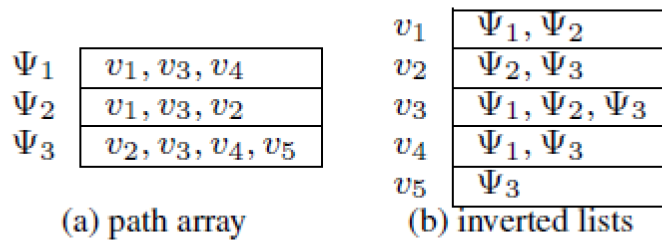
1: $\Psi \leftarrow$ new cache;
2: $H \leftarrow$ new max-heap; ▷ storing result paths
3: **for each** $Q_{a,b} \in \mathcal{QL}$ **do**
4: $\quad P_{a,b}.\Delta\overline{\gamma} \leftarrow \Delta\overline{\gamma}(P_{a,b}, \Psi)$; ▷ compute using χ and E
5: $\quad$ insert $P_{a,b}$ into H;
6: **while** $|\Psi| \leq \mathcal{B}$ and $|H| > 0$ **do**
7: $\quad P_{a',b'} \leftarrow H.pop()$; ▷ potential best path
8: $\quad P_{a',b'}.\Delta\overline{\gamma} \leftarrow \Delta\overline{\gamma}(P_{a',b'}, \Psi)$; ▷ update $\Delta\overline{\gamma}$ value
9: $\quad$ **if** $P_{a',b'}.\Delta\overline{\gamma} \geq$ top $\Delta\overline{\gamma}$ of H **then** ▷ actual best path
10: $\quad\quad$ **if** $\mathcal{B} - |\Psi| \geq |P_{a',b'}|$ **then** ▷ enough space
11: $\quad\quad\quad$ insert $P_{a',b'}$ into Ψ;
12: $\quad$ **else** ▷ not the best path
13: $\quad\quad$ insert $P_{a',b'}$ into H;
14: return Ψ;

1. Algoritmus

Gyorsítótár struktúra

Ezek után meg kell terveznünk egy gyorsítótár struktúrát, amely hatékony elérési időt biztosít számunkra a lekérdezésekhez.

A gyorsítótár lekérdezésére egy hatékony módszer a fordított listák használata. Ennek a struktúrának a használata során felhasználunk egy útvonaltömböt és egy csúcslistát (ahogy a 6. Példán láthatjuk). A tömbben eltároljuk az összes útvonal tartalmát. Ha van egy $Q_{s,t}$ lekérdezésünk, akkor csak meg kell vizsgálni v_s és v_t-hez tartozó fordított listát. Ha ennek a két listának a metszete nem üres, akkor biztosak lehetünk benne, hogy Ψ_j-hez tartozó útvonal megválaszolja a $Q_{s,t}$ lekérdezést.



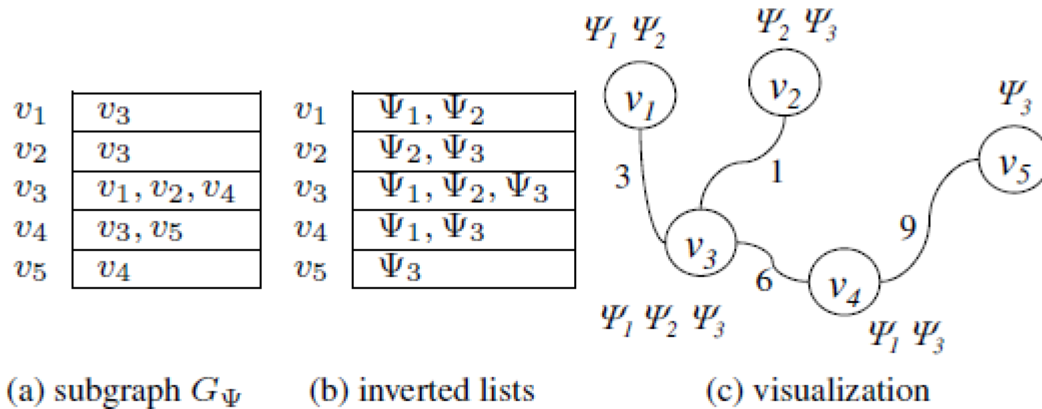
6. Példa: Fordított listák struktúra

Méretelemzés: Legyen |Ψ| a csúcsok száma, és legyen m az útvonalak száma az útvonaltömbünkben. Vegyük észre, hogy egy x méretű bejegyzés egy $I_x = |\log_2 x|$ méretű bináris karaktersorozattal tárolunk.

Az útvonaltömbben minden csúcs reprezentálható $I_{|V|}$ bittel. Így az útvonaltömb $|\Psi| * I_{|V|}$ bitet foglal. Minden fordított listában, minden útvonal azonosító reprezentálható I_m bittel.

Megjegyezzük, hogy az útvonal azonosítók száma a fordított listákban |Ψ|-vel egyenlő. Így a fordított listát $|\Psi| * I_m$ bit helyet foglalnak. Tehát a teljes struktúra mérete: $|\Psi| * (I_{|V|} + I_m)$ bit.

Egy másik struktúra amit alkalmazhatunk a gyorsítótárhoz az egy részgráf modell használata. A módszer nagyon hasonló az előző módszerhez, annyi különbséggel, hogy az útvonaltömb helyett egy GΨ gráf tömbös reprezentációját alkalmazzuk. A megvalósítását és annak vizualizációját a következő példán láthatjuk:

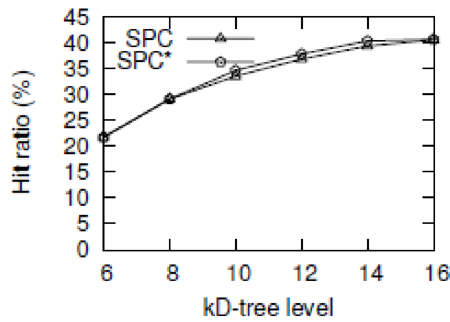


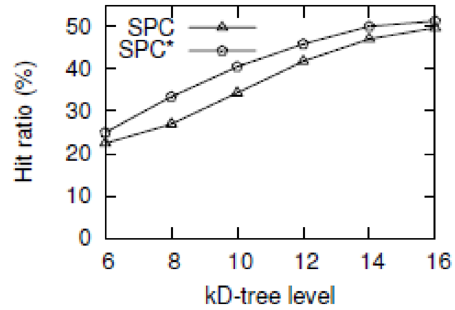
7. Példa: Részgráf modell struktúra

Méretelemzés: Legyen |Ψ| a csúcsok száma és m az útvonalak száma a gyorsítótárban, V_Ψ részhalmaza V pedig a külön osztályba tartozó csúcsok halmaza, e pedig a csúcsonkénti átlagos szomszédszám. A fordított listák $|\Psi| * I_m$ helyet foglalnak, ahogy az előzőekben láthattuk. A részgráf $|V_\Psi| * e * I_{|V|}$ helyet foglal, így összesen a struktúra $|V_\Psi| * e * I_{|V|} + |\Psi| * I_m$ bitet foglal.

A megkapott algoritmusunkat és gyorsítótár struktúránkat ezek után letesztelhetjük összevetve más gyorsítótár használati módszerekkel. Az az előzőekben említett LRU és HQF módszereket vetjük össze az általunk előállított SPC (Shortest-Path-Cache) azaz legrövidebb útvonal gyorsítótár módszerrel, majd az SPC* módszerrel amely a kompakt struktúrát használja az elsőnek definiált fordított listák helyett. Tesztjeink során két valós adathalmazzal fogunk dolgozni: Aalborg és Peking úthálózataival.

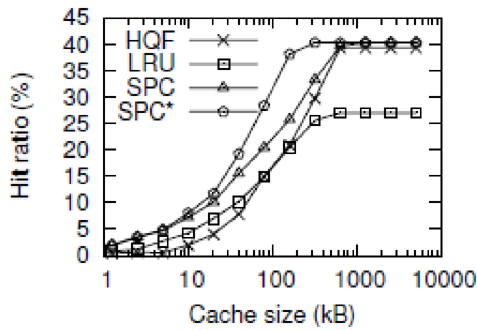
Első lépésként teszteljük a proxy szerveren elhelyezett gyorsítótár esetén az előállított kD-fa magasságánál mekkora különbségeket találunk SPC és az SPC* módszerek között. Minél magasabb a kD-fa annál pontosabb lekérdezési statisztikákat tartalmaz így megnövelve a gyorsítótár haszon mutatótáját. Ahogy láthatjuk mindkét adathalmaz esetén az SPC* jobb eredményeket produkál.

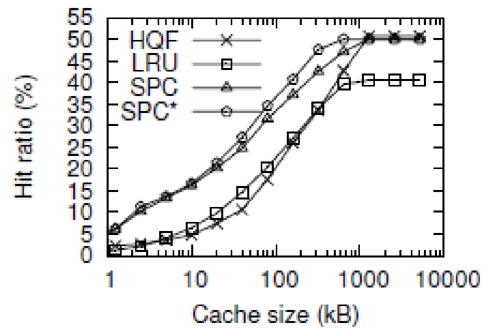




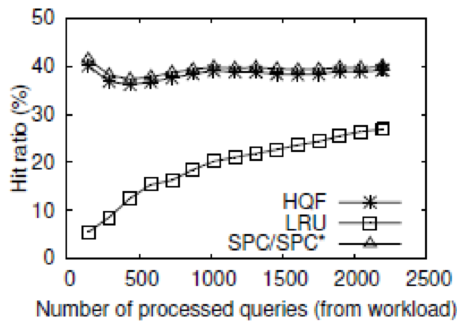
11. Példa: Találati arány vs. szintek

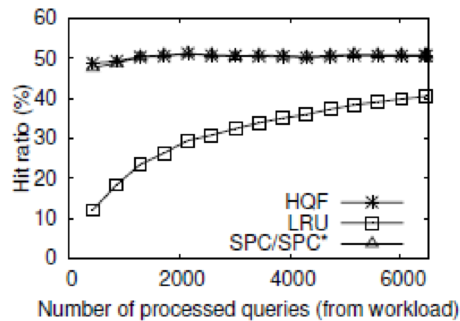
Következő tesztünkben a gyorsítótár találati arányokat fogjuk összehasonlítani a gyorsítótár méretének függvényében a meglévő módszerekkel, ahogy azt a 12. Példán láthatjuk. 13-as példán pedig a feldolgozott lekérdezések arányában vett gyorsítótári találatokat hasonlítjuk össze. Amint láthatjuk az SPC* még mindig a legjobb eredményt éri el minden esetben.





12. Példa: Találati arány vs. Cache méret

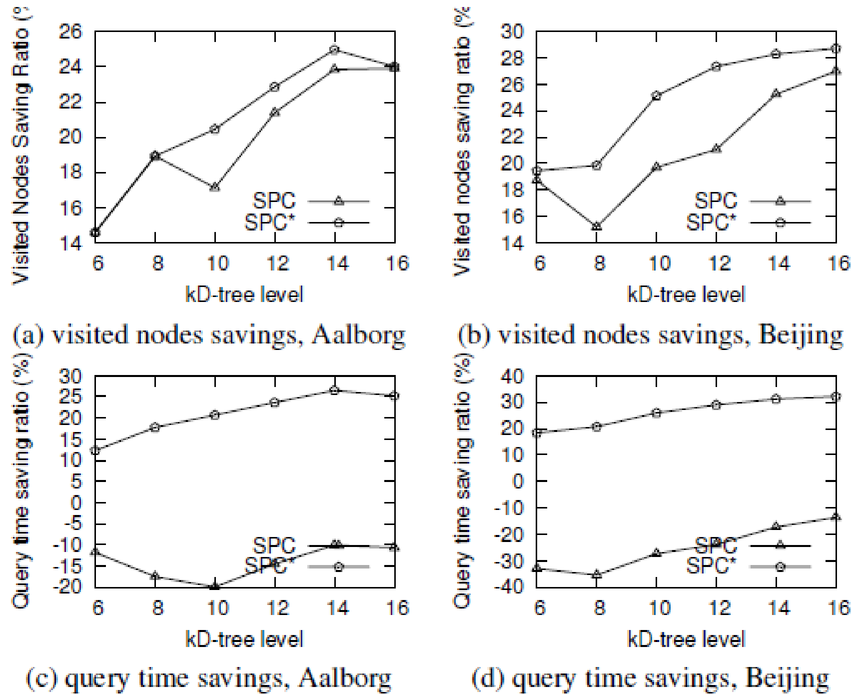




13. Példa: Találati arány vs. feldolgozott lekérdezések (5 Mbyte-os cache esetén)

Megjegyezhetjük, hogy a módszereink viszonylag gyorsan felépítik a gyorsítótárat még ekkora adatok esetén is. Az SPC és SPC* kevesebb, mint 3 perc alatt építette fel mind Aalborg és Peking adatainak segítségével gyorsítótárát. Mivel SPC adatstruktúrája egyszerűbb így annak felépítése fele annyi időbe telt, mint SPC* esetén.

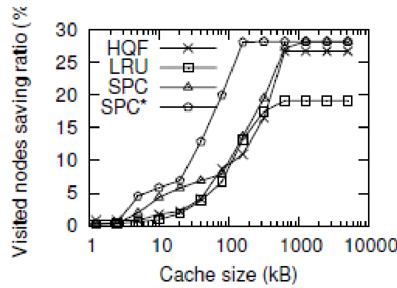
SPC és SPC* módszereket ezek után lefuttattuk a szerver oldali tárolás módszerével is. Először megvizsgáltuk mennyi teljesítménynövekedést értünk el a kD-fa méretének növekedésével az A* és Dijkstra algoritmusok esetén.



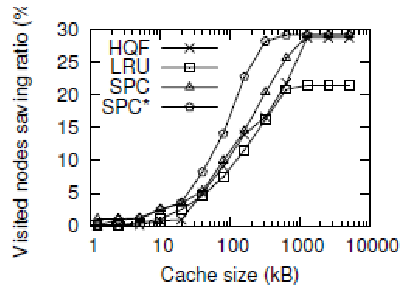
(a) visited nodes savings, Aalborg (b) visited nodes savings, Beijing

(c) query time savings, Aalborg (d) query time savings, Beijing

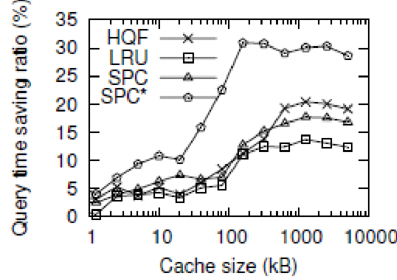
Ezután megvizsgáltuk a teljesítménynövekedést Dijkstra és A* esetén a gyorsítótár méretének növekedése esetén. A következő eredményeket kaptuk:



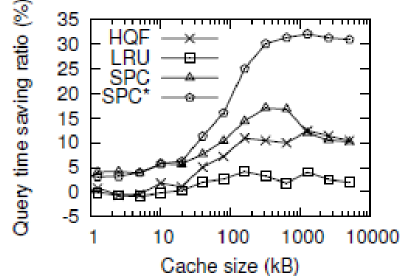
(a) visited nodes savings, Aalborg



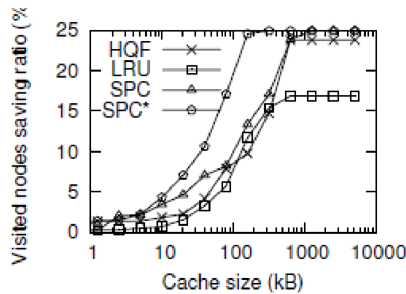
(b) visited nodes savings, Beijing



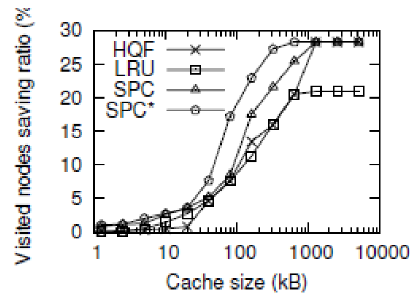
(c) query time savings, Aalborg



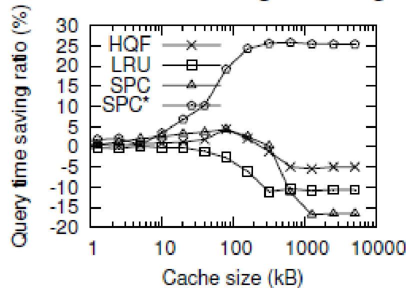
(d) query time savings, Beijing



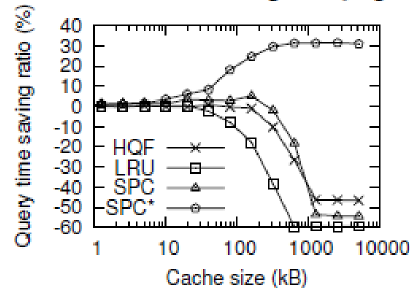
(a) visited nodes savings, Aalborg



(b) visited nodes savings, Beijing



(c) query time savings, Aalborg



(d) query time savings, Beijing

További kutatási terv

Tanulmányunkban a proxy és szerver oldali legrövidebb útvonalak gyorsítótárral történő eltárolását vizsgáltuk. Létrehoztunk egy modellt mellyel meg tudtuk határozni egy útvonal eltárolásának hasznosságát annak gyakorisága és feldolgozása költségének függvényében. Kifejlesztettünk olyan technikákat, melyek segítségével kiszámolhattuk a lekérdezések

gyakoriságának statisztikáit és megbecsülhettük a legrövidebb útvonalat kiszámító algoritmus költségét. Egy mohó algoritmust alkalmaztunk, amellyel kiválasztottuk a legígéretesebb útvonalakat számunkra egy lekérdezési naplóból, amelyeket eltároltunk a gyorsítóárban. Ezen felül megadtunk több gyorsítótár struktúrát a lekérési költségek optimalizálására és helyfoglalás minimalizálására. A tesztjeink azt mutatják, hogy valós adatokkal a legjobb módszerünk az SPC* produkálta a legjobb eredményt mind a proxy, mind a szerver oldali tárolás esetén.

Amit nem vizsgáltunk az a meghívott útvonalkereső API optimalizálása a saját gyorsítótár adatstruktúránk számára. A vizsgált A* és Dijkstra algoritmusok helyett felhasználhatnánk egy saját heurisztikus keresőalgoritmust, amely figyelembe veszi a használt gyorsítótár struktúránkat és annak elemeit.

Irodalomjegyzék:

C. S. Jensen was supported in part by the EU STREP project, Reduction.

We thank Eric Lo, Jianguo Wang, Yu Li, and the anonymous

reviewers for their insightful comments.

[1] I. S. Altingövde, R. Ozcan, and Ö. Ulusoy. A Cost-Aware Strategy for Query Result Caching in Web Search Engines. In ECIR, pp. 628–636, 2009.

[2] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The Impact of Caching on Search Engines. In SIGIR, pp. 183–190, 2007.

[3] R. A. Baeza-Yates and F. Saint-Jean. A Three Level Search Engine Index Based in Query Log Distribution. In SPIRE, pp. 56–65, 2003.

[4] T. H. Cormen, C. E. Leiserson, and C. Stein. Introduction to Algorithms. MIT Press, 3rd edition, 2009.

[5] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In VLDB, pp. 330–341, 1996.

[6] Q. Gan and T. Suel. Improved Techniques for Result Caching in Web Search Engines. In WWW, pp. 431–440, 2009.

[7] M. C. González, C. A. Hidalgo, and A. L. Barabási. Understanding Individual Human Mobility Patterns. Nature, 453(7196):779–782, 2008.

[8] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. In SIGMOD, pages 205–216, 1996.

[9] H. Hu, D. L. Lee, and V. C. S. Lee. Distance Indexing on Road Networks. In VLDB, pp. 894–905, 2006.

[10] H. Hu, J. Xu, W. S. Wong, B. Zheng, D. L. Lee, and W.-C. Lee. Proactive Caching for Spatial Queries in Mobile Environments. In ICDE, pp. 403–414, 2005.

[11] C. S. Jensen, H. Lahrmann, S. Pakalnis, and J. Runge. The Infati Data. CoRR, cs.DB/0410001, 2004.

[12] S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. IEEE TKDE, 14(5):1029–1046, 2002.

[13] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt. Hierarchical Graph Embedding for Efficient Query Processing in Very Large Traffic Networks. In SSDBM, pp. 150–167, 2008.

[14] K. Lee, W.-C. Lee, B. Zheng, and J. Xu. Caching Complementary Space for Location-Based Services. In EDBT, pp. 1020–1038. 2006.

[15] X. Long and T. Suel. Three-Level Caching for Efficient Query Processing in Large Web Search Engines. In WWW, pp. 257–266, 2005.

[16] E. P. Markatos. On Caching Search Engine Query Results. Computer Communications, 24(2):137–143, 2001.

[17] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy. A Five-Level Static Cache Architecture for Web Search Engines. Information Processing & Management, 2011.

[18] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Static Query Result Caching Revisited. In WWW, pp. 1169–1170, 2008.

[19] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast Shortest Path Distance Estimation in Large Networks. In CIKM, pp. 867–876, 2009.

[20] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable Network Distance Browsing in Spatial Databases. In SIGMOD, pp. 43–54, 2008.

[21] F. Wei. TEDI: Efficient Shortest Path Query Answering on Graphs. In SIGMOD, pp. 99–110, 2010.

[22] B. Zheng and D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In SSTD, pp. 97–116, 2001.

[23] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining Interesting Locations and Travel Sequences