Eötvös Loránd Tudományegyetem

Természettudományi Kar

Tóth Lilla

Útvonal optimalizálás a gyakorlatban lokális kereséssel

BSc Szakdolgozat

Témavezető:

Szabó Csaba

Algebra és Számelmélet Tanszék

Budapest, 2013

Tartalomjegyzék

1. Egy sütőipari cég szállítási problémája **3**

2. Lokális keresés az útvonaltervezésben és az ütemezési problémákban **5**

2.1. Bevezetés, célmeghatározás 5

2.2. Lokális keresés 6

2.2.1. A lokális keresés algoritmusa 7

2.2.2. Lépések és felbontásaik 8

2.2.3. Az útvonaltervezés és ütemezési probléma megoldásának reprezentálása 9

2.2.4. Korlátok és megoldhatóság 10

2.2.5. Egy (rész)lépés költsége és haszna 10

2.3. Szomszédsági típusok 11

2.3.1. Csúcs-cserélő szomszédságok 12

2.3.2. Él-cserélő szomszédságok 13

2.4. Keresési technikák 17

2.4.1. Közvetlen keresés 18

2.4.2. Optimalizálás alapú, indirekt keresési technikák 21

3. A probléma megoldása a TransIT nevű szoftverrel **24**

1. fejezet

Egy sütőipari cég szállítási problémája

Útvonal optimalizálást a gyakorlatban sok helyen alkalmaznak. Ebben a fejezetben egy konkrét problémát mutatunk be, amely egy sütőipari termékeket szállító cég életében merült fel.

A mai labilis gazdasági környezetben, a dráguló üzemanyagárak mellett egyre nagyobb fontossággal bír, hogy egy cég az erőforrásait a lehető legoptimálisabban használja ki annak érdekében, hogy felvehesse a versenyt a piacon a konkurenciával.

Ebben a példában az a cél, hogy egy adott napra leadott összes rendelést teljesítsük, azaz elszállítsuk oda a megrendelt árut. Mindezt úgy tegyük meg, hogy az összes teherautó által megtett összes kilométer minimális legyen. Emellett fontos szempont az is, hogy a személyi munkaerőt is optimálisan használjuk, azaz lehetőleg ne lépjük túl a 8 órás munkaidőt.

A feladat megoldását egy munkanap útvonaltervének megadásával szimuláltuk. Adva volt 127 megrendelés/bolt, ahol a következő korlátoknak/paramétereknek kellett megfelelni:

- Meg volt adva minden megrendelés pontos címe: irányítószám, város, utca, házszám.
- Minden boltnál meg volt adva, hogy milyen idősávban van az áruátvétel, amikor ki lehet szállítani az árut.
- Meg volt adva a megrendelt áru mennyisége kilogrammban.
- Tudtuk, hogy egy boltban mennyi idő szükséges a kipakoláshoz, azaz mennyi időt szükséges az adott boltban tölteni.

A cég egy depóval rendelkezik, így minden teherautó onnan indul és oda is kell visszatérnie. A depónak ismert a pontos címe.

A szállítás elvégzéséhez 21 db teherautó állt rendelkezésünkre, melyekről az alábbi korlátokat/paramétereket tudtuk:

- Minden teherautónak adott a súlya.
- Tudjuk a maximális terhelhetőségüket kilogrammban.
- Meg van adva, hogy melyik autónak milyen területekre van behajtási engedélye. Pl.: melyik teherautónak van autópálya matricája, vagy melyik hajthat be a Budai Várba.

A gyakorlatban, optimalizálási eszköz használata nélkül, ezt a feladatot 19 teherautóval oldották meg, amelyek összesen 3000 kilométert tettek meg. Ezt az eredményt kellett javítanunk a TransIT szoftver használatával.

A dolgozatban betekintést adunk a szoftver működéséhez szükséges matematikai háttérbe és a felhasználói felületbe. Azonban nem törekszünk a tejleskörű leírásra, mivel az esetenként túlmutatna a BSc-s szakdolgozat keretén, máskor pedig túlzottan technikai jellegű lenne. Így csak nagy vonalak mentén tekintjük át a lokális keresés működését, és az ehhez szükséges fogalmakat. Definiáljuk a lépés és a szomszédság fogalmát, majd bemutatunk néhány szomszédságot, és pár alapvető keresési technikát.
Majd a harmadik fejezetben részletesen bemutatjuk, hogy a TransIT szoftverben hogyan oldottuk meg az előbbiekben leírt feladatot.

2. fejezet

Lokális keresés az útvonaltervezésben és az ütemezési problémákban

2.1. Bevezetés, célmeghatározás

Az útvonal keresési probléma egy optimalizálási feladat, ahol adott egy jármű állomány, amelynek előre megadott megrendeléseket kell teljesítenie úgy, hogy az összesen megtett kilométer minimális legyen. A feladatot ki lehet egészíteni ütemezési feladattá, ha megadunk egy idősávot minden megrendeléshez, és kikötjük, hogy csak ebben az idősávban lehet a megrendelést teljesíteni. Az útvonal keresési probléma megoldásaként meg kell adnunk egy útvonaltervet, aminek tartalmaznia kell, hogy melyik járművel, mely megrendeléseket, milyen sorrendben teljesítjük. Az ütemezési feladat megoldásánál pedig meg kell mondani, hogy mikor érkezünk egy adott megrendeléshez és mikor távozunk onnan.

Jelenleg nem ismerünk gyors algoritmust nagyméretű útvonal tervezési és ütemezési problémák megoldására, mert a feladat $\mathcal{NP}$-teljes. Ezért az eddig legjobbnak bizonyuló lokális keresést fogjuk bemutatni. A következő fejezetekben a klasszikus és a modern lokális keresési technikákról lesz szó.

Az útvonal keresési és ütemezési feladatok megoldását az úgynevezett óriástúra modellel fogjuk elemezni, és osztályozzuk a megoldások szomszédsági struktúráját. A szomszédságokat úgy adjuk meg, hogy megmondjuk a lépések halmazát, amik az egyik megoldást a másikba transzformálják. Majd megmutatjuk, hogyan lehet a lépéseket részlépésekre bontani, és ezek után a keresési technika hogyan illeszti ezeket össze egy lépéssé hatékony módon.

A cél, hogy találjunk egy lokálisan optimális szomszédot olyan gyorsan, amilyen gyorsan csak lehetséges. Ezt egy olyan keresési technikával érjük el, amely nem nézi végig explicit módon az összes szomszédot. Az elemzés során látni fogjuk, hogy a részlépések tulajdonságai és az útvonal tervezési illetve ütemezési problémák megkötései hogyan befolyásolják a megfelelő keresési technika megválasztását.

Az útvonal tervezési és ütemezési problémákat két fő szempontból fogjuk megvizsgálni:

1. A szomszédság választása.

2. A szomszédság felderítése megfelelő algoritmussal.

A keresési módszerek és a szomszédsági struktúrák között szoros kapcsolat van, melyet sok algoritmus használ (például a Lin-Kernighan algoritmus, ami megoldja az utazó ügynök problémát). Azonban, az egyik legnagyobb kihívás, különösen az összetett és nagyméretű problémáknál, hogy úgy tervezzük meg a metaheurisztikát, hogy gyorsan lefusson. Ezt elősegíti a szomszédság és a keresési technika megfelelő megtervezése.

Kiemelkedő példák útvonal keresési problémára:

- Utazó ügynök probléma (Travel Salesman Problem) [Lawler 1985., Gutin és Punnen2002.]

- Jármű útvonal tervezési probléma (Vehicle Routing Problem) [Tóth és Vigó 2002.]

- Felvételi és kiszállítási probléma (Pickup- and Delivery Problem) [Salvelsbergh és Sol 1985., Desaulniers 2002.]

2.2. Lokális keresés

Ebben a fejezetben ismertetjük a lokális keresés alapjait. Először bevezetünk pár jelölést.

Adott egy $G = (V, A)$ gráf, aminek pontjai az előző problémában a megrendelések és a depó földrajzi helyét jelölik. Éleket pedig oda húzunk be, amely két földrajzi hely között lehet közúton vagy komppal közlekedni. Az éleken adott egy $c : A \to \mathbb{R}$ költségfüggvény, ez az útszakaszon való haladás költsége. Ide tartozik például az üzemanyagköltség vagy az útdíj.

Emellett egyéb korlátokat is figyelembe kell vennünk, például a járművek kapacitását, behajtási engedélyét, a nyitvatartási időket.

Az x megoldása az útvonal keresési problémának, ha G-beli Hamilton-kör, vagy Hamilton-körök uniója. Az x megoldás megengedett, ha megoldás és nem sért meg egy korlátot sem. Legyen X a megengedett megoldások halmaza. Ez a halmaz nagyméretű, de véges. A továbbiakban a költségfüggvényt nem az éleken értelmezzük, hanem egy megoldásnak fogjuk tekinteni az összköltségét, így a költségfüggvény a megoldásokhoz rendel valós számokat ($c : X \to \mathbb{R}$). Optimális megoldásnak nevezzük az $x \in X$ megoldást, ha teljesül, hogy $\{\forall x' \in X : c(x) \leq c(x')\}$. Az optimális megoldás megtalálása egy kombinatorikus optimalizálási feladat.

A lokális keresés alapművelete a lépés, ami egy x (nem feltétlenül) megengedett megoldást egy másik (nem feltétlenül) megengedett megoldásba transzformál élek és/vagy csúcsok cseréjével. Az x megoldás szomszédjának vagy szomszédos megoldásának nevezzük az x' megoldást, ha x'-t meg lehet kapni x-ből egy lépés alkalmazásával. Az x megoldás szomszédos megoldásainak halmazát az x megoldás szomszédságának nevezzük,és $N(x)$-szel jelöljük.

Egy x' megoldást javító megoldásnak hívunk, ha $x' \in N(x)$ és $c(x') < c(x)$. Ha x megengedett megoldásnak nincs szomszédos javító megoldása, akkor lokális optimumnak nevezzük. Ekkor $\forall x' \in N(x) : c(x') \geq c(x)$.

2.2.1. A lokális keresés algoritmusa

A lokális keresés algoritmusának lépései a következők:

1. Az algoritmus inputként kap egy $x^0 \in X$ kezdeti megoldást, továbbá az iteráció számlálóját beállítjuk nullára, azaz $t := 0$

2. Ismételjük az alábbi ciklust:

3. Keressünk egy javító szomszédot $N(x^t)$-ben az aktuális megoldáshoz!

4. Ha létezik ilyen x', akkor $x^{t+1} := x'$ és $t := t + 1$

5. Leállunk, ha nincs javító szomszéd, mert ez azt jelenti, hogy lokális optimumot kaptunk.

A szomszédságban való keresés gyorsasága nagymértékben függ attól, hogy milyen gyorsan tudjuk ellenőrizni az x' szomszédos megoldás költségét és megengedettségét.

Mivel a 3. lépésben nem kell az összes javító szomszédot megkeresni, hanem elég egy javító megoldást megtalálni, az algoritmus futási ideje ezzel nagy mértékben csökken ahhoz képest, ha az összes szomszédos megoldást végig kéne nézni. Ha a szomszédos megoldások halmaza felsorolható, akkor az alábbi stratégiák közül választhatunk:

1. First search: Megkeresi az első javító megoldást.

2. The best search: Megtalálja a legjobb javító megoldást.

3. d-best search: Megvárja, amíg d db javító megoldást megtalál, és abból választja ki a legjobbat.

Azonban az utolsó (5.) lépésben mindenképpen végig kell nézni a lokálisan legjobb megoldás összes szomszédos megoldását ahhoz, hogy igazoljuk az optimalitást. Mert az algoritmus csak ebben az esetben állhat le.

Ha egy adott szomszédságban meg szeretnénk találni a legjobb javító szomszédot, akkor ahhoz szintén egy optimalizálási problémát kell megoldanunk. Erre többféle megoldási módszer is van, pl.: explicit leszámlálási technika vagy egy alkalmas optimalizálási algoritmus (dinamikus programozás, hálózati optimalizálás, legrövidebb utat megtaláló algoritmus).

A szomszédság dinamikus is lehet, azaz $N(x^t)$ függhet az iteráció számától és az előző keresésektől.

2.2.2. Lépések és felbontásaik

Ahogy a lokális keresés bevezetésében említettük, a lépés egy művelet, ami megoldást megoldásba transzformál élek és/vagy csúcsok cseréjével. Jelöljük M-mel a lépések halmazát, amelynek elemeit, azaz a lépéseket, pedig jelölje m. Azért, hogy együtt tudjuk kezelni a megengedett és nem megengedett megoldásokat, vezessük be $Z \supseteq X$ halmazt, amire $y \in Z$, ha $\exists m \in M$ lépés, ami egy megengedett megoldást egy $y = m(x)$ nem feltétlenül megengedett megoldásba visz.

Mivel a lépés eredménye nem mindig megengedett megoldás, ezért $m : Z \to Z$ nem mindig jól definiált, mivel nem feltétlen alkalmazható minden $y \in Z$-re. Azonban, ha $m \in M : m(x) \subset X$, akkor m-et megengedett lépésnek hívjuk. Azért, hogy kezelni tudjuk a nem megengedett szomszéd megoldásokat is, vezessük be a kiterjesztett szomszédság fogalmát: $\widehat{N}(x) := \{m(x) : m \in M\}$. Ez a halmaz már tartalmazza a megengedett és nem megengedett szomszédokat is. A kiterjesztett szomszédság és a megengedett megoldásokat tartalmazó szomszédság között a következő kapcsolat áll fenn: $N(x) = \widehat{N}(x) \cap X$.

A szomszéd megoldások számát a szomszédság méretének nevezzük. Mivel $|N(x)|$ általában függ x-től, többnyire $\widehat{N}$ méretét fogjuk vizsgálni.

Ahhoz, hogy a különböző lépéseket elemezni tudjuk, a lépést részlépésekre bontjuk. A

részlépés fogalmát nem definiáljuk pontosan, mivel az túlságosan technikai jellegű lenne. Részlépés alatt általában él(ek) és/vagy csúcs(ok) eltávolítását vagy beillesztését értjük az aktuális megoldásból/megoldásba. Formailag ez így írható: $m(x) := p_\ell \circ \cdots \circ p_2 \circ p_1 = p_\ell(\ldots p_2(p_1(x)))$. A részlépésekre bontás nem nyilvánvaló és nem egyértelmű. Így természetesen adódik a kérdés, hogy mit lehet mondani a részlépések által generált köztes struktúrákról: $m : Z = Y_0 \xrightarrow{p_1} Y_1 \xrightarrow{p_2} \ldots \xrightarrow{p_l} Y_\ell = Z$, ahol az Y_i-ket hívjuk köztes struktúráknak. Az Y_i-k lehetnek Z-től különbözőek, például ahogy a Lin-Kernighan szomszédságnál látjuk majd, de akár végig megegyezhetnek Z-vel.

2.2.3. Az útvonaltervezés és ütemezési probléma megoldásának reprezentálása

Ahogy a bevezetésben említettük, a megoldást egy irányított gráffal fogjuk reprezentálni, melynek csúcshalmazán belül a következő típusú csúcsokat különböztetjük meg:

- Követelményt támasztó csúcsok: $R \subset V$
- Lehetséges kezdőpontok: $O \subset V$
- Lehetséges végpontok: $D \subset V$

Azt, hogy a követelményt támasztó csúcs alatt mit értünk, mindig az aktuális probléma határozza meg. Például útvonal tervezési problémánál ezek azok a csúcsok, amelyeknek megfelelő megrendeléseket mindenképpen szeretnénk teljesíteni. Ha nem csak kiszállítunk, hanem ott fel is kell vennünk csomagot, amit tovább kell szállítani, akkor a követelmény az, hogy vigyük el a boltba a megrendelt árut, és szállítsuk tovább az általuk küldött csomagot. Ha ütemezési problémáról van szó, akkor az a követelmény, hogy az adott boltot a megadott idősávban látogassuk meg.

Ahogy az elején is említettük, a megoldás vagy egy Hamilton kör, vagy Hamilton-körök uniója. Ezt utakból álló, H-hosszú listaként kezeljük, és útvonaltervnek hívjuk: $X = (r^1, r^2, \ldots, r^H)$. Mivel az utak Hamilton-kört alkotnak, ebből adódóan diszjunktak, azaz $\forall v \in V$ pontosan egyszer szerepel valamely $r^i - ben$. A továbbiakban feltesszük, hogy $|V| = n$ és az r^i által lefedett csúcsok halmaza: $V(r^i)$.

A megoldásban olyan utak vannak, melyek $o \in O$-ból indulnak és $d \in D$-be érkeznek, közben pedig $r \in R$ követelményt támasztó pontokat látogatnak meg, ahol minden korlátnak megfelelnek. Az, hogy egy (o, d) pár megengedett-e, szintén a problémától függ.

Még egy fontos definíciót kell bevezetnünk: óriástúrának fogjuk hívni az útvonaltervet, ha $\forall i$-re r^i végpontja egyenlő r^{i+1} kezdőpontjával.

2.2.4. Korlátok és megoldhatóság

Az útvonal tervezési és ütemezési feladat megoldhatóságára vonatkozó korlátokat úgynevezett erőforrás változókkal lehet modellezni. Ezen belül megkülönböztetünk egy útra vonatkozó és utak közötti korlátokat. Útra vonatkozó korlát lehet például a járművek kapacitása, a megrendelések közötti prioritás vagy a rendeléseknél megadott idősáv. Útvonalak közötti korlát például, ha megszabjuk, hogy csak n db k csúcsnál hosszabb utat tartalmazhat az óriástúra, vagy ha a depóban egyszerre csak korlátozott számú jármű tartózkodhat. Jelenleg nem tudunk olyan szisztematikus megközelítésről, ami megfelelően kezelné az útvonalak közötti korlátokat, habár néhány alkalmazásban jelentős szerepet játszanak. Ebből adódóan a következőkben az útra vonatkozó korlátokat fogjuk vizsgálni.

A korlátozott erőforrásokat formálisan bemutatni az idősáv példáján lehet a legegyszerűbben. Legyen $T_i, i \in V$ a szolgálati idő kezdete, amikortól a járművek elindulhatnak. Ekkor ha o^i az i. kezdőpont, akkor T_{o^i} az indulási idő. A legtöbb ütemezési problémában meg van adva minden csúcsra egy vagy több idősáv: $[a_i^\ell, b_i^\ell]$, $\ell = 1, 2, \ldots, L(i)$ $\forall i \in V$, ahol $L(i)$ az i. csúcson megadott utolsó idősáv sorszáma. Ezek az intervallumok adnak feltételeket a megoldhatóságra, mivel az adott megrendelést csak ezekben az idősávokban lehet teljesíteni. Ha egy jármű i-ből j-be megy, az idő legalább $t_{ij} + s_i$-vel nő, ami az utazási idő i-ből j-be, plusz a kiszolgálási idő i-ben. Ha kiválasztottuk az (i, j) élt, az azt jelenti, hogy $T_j - T_i \geq t_{ij} + s_i$, tehát gyorsabban értünk j-be, mintha másik úton mentünk volna.

Az útvonalterv megengedettségét lehet ellenőrizni egy $\widehat{o}$ orákulummal, ami egy $\widehat{o} : Z \to \{igen, nem\}$ logikai függvény. Ekkor $N(x) = \{m(x) : m \in M, \widehat{o} = igen\}$. Azonban az orákulum meghívása nagyban csökkenti a hatékonyságot, ezért óvatosan kell vele bánni.

2.2.5. Egy (rész)lépés költsége és haszna

Egy $x \in Z$ óriástúra költsége az óriástúrában lévő élek összköltsége ($c(x) := \sum_{(i,j) \in \text{gianttour(x)}} c_{ij}$). Egy részlépés haszna az a szám, amennyivel a részlépés csökkentette az összköltséget ($g(m, x) := c(x) - c(m(x))$).

Az alábbi három esetet használja majdnem az összes releváns alkalmazás:

1. Élfüggő költség: $c_{ij}(x) = c_{ij}$, $\forall(i,j) \in A$. Ez a legegyszerűbb, mivel a költség csak az éltől függ, így mind a haszon, mind a költség egyszerűen és gyorsan számolható. Ezt használja például az utazó ügynök probléma is.

2. Járműfüggő költség: $c_{ij}(x) = c_{ij}^{\mathrm{od}}$, minden (i,j) élre, amit az o kezdőpontú és d végpontú út tartalmaz. Ez a költség függ a kiindulási és az érkezési ponttól. Így a viszonylagos haszon csak akkor meghatározható, ha az utakban előre kijelöljük a követelményt támasztó pontokat. Ezt például a multi-depó problémánál alkalmazzuk, ahol több depóból szállítjuk ki az árut.

3. Erőforrásfüggő költség: $c_{ij}(x) = c_{ij}(T_i^1, \dots, T_i^p, T_j^1, \dots, T_j^p)$, ahol $(T_i^1, \dots, T_i^p, T_j^1, \dots, T_j^p)$ erőforrás változók, $i, j \in V$. Ezzel lehet jól modellezni az idő- és rakományfüggő költségeket.

Ahhoz, hogy a lokális keresés hatékony legyen szükséges, hogy minden lépés részlépésre bontásakor, annak $g(p_i, x)$ hasznát meghatározzuk, ami csak az aktuális megoldástól függ ($m \in M : m = p_\ell \circ \dots \circ p_2 \circ p_1$). Fontos, hogy a részlépésekre bontással ne változzon az összhaszon, ha ez teljesül: $g(m, x) = \sum_{i=1}^{\ell} g(p_i, x)$, akkor a részlépésre bontást költségfüggetlennek nevezzük. Ha viszont nem minden esetben teljesül, akkor megadhatunk úgynevezett "jogosultsági feltételeket", amik szükséges feltételt adnak a függetlenségre. Például rögzíthetjük a részlépések sorrendjét.

Abban az esetben, ha $Y = Z$, az $m = p_\ell \circ \dots \circ p_2 \circ p_1$ lépést sorrendfüggetlennek hívjuk, ha $m(x) = p_{\pi(\ell)} \circ \dots \circ p_{\pi(2)} \circ p_{\pi(1)}$, $\forall x \in Z$ megoldásra és $\forall \pi \in \{1, 2, \dots, \ell\}$ permutációra. Továbbá ciklikusan függetlennek hívjuk, ha az előbbi feltétel csak a ciklikus permutációkra teljesül.

Ha a pontos kiszámítás túl sokáig tartana, akkor közelítéssel és alsó becsléssel is lehet gyorsítani a keresési folyamatot.

2.3. Szomszédsági típusok

Ebben a fejezetben bemutatunk pár lehetséges szomszédsági típust. A válogatási szempont az volt, hogy a később tárgyalt keresési technikák által használt szomszédsági típusokat bemutassuk.

A továbbiakban feltesszük, hogy adott egy x megoldása az útvonal tervezési feladatnak. Továbbá feltesszük, hogy bármely két megoldás egymásba vihető élek hozzáadásával és

elvételével.

Kétféle szomszédságról lesz szó, a csúcs-cserélő és az él-cserélő szomszédságról, amelyek mindegyikére mutatunk két-két példát.

2.3.1. Csúcs-cserélő szomszédságok

A csúcs-cserélő szomszédság arról kapta a nevét, hogy a szomszédságot generáló lépést egyszerűbb leírni a résztvevő csúcsokkal.

Tegyük fel, hogy adott ℓ diszjunkt szegmens az útvonaltervből $s_1, s_2, \ldots, s_\ell$, legyen $|s_i|$ a csúcsok száma s_i szegmensben. Ekkor a csúcs cserélés leírható egy ℓ dimenziós $\alpha = (\alpha_1, \ldots, \alpha_\ell)$ vektorral, ahol α_i ($\alpha_i \leq |s_i|$) csúcsot helyezünk át az s_i szegmensből s_{i+1} szegmensbe (ha $i = \ell$, akkor s_ℓ szegmensből s_1 szegmensbe).
Ha feltesszük, hogy a beillesztendő csúcsokat már törölt csúcsok helyére teszünk be, akkor α_i helyett $\underline{\alpha_i}$-t írunk. Ha pedig α_i-nél kevesebb csúcsot is elég kiválasztani, akkor α_i helyett α_i^*-t fogjuk használni.

Kicserélés

A kicseréléskor két út egy-egy csúcsát cseréljük ki. Ennek leírásához a két útnak összesen 10 csúcsára van szükség, legyenek ezek a következők $\{i_1-1, i_1, i_1+1, j_1, j_1+1, i_2, i_2+1, j_2-1, j_2, j_2+1\}$, ahol az i_k és j_ℓ csúcsok vannak egy-egy útban. A lépést négy részlépésre bontjuk. Az első részlépésben eltávolítjuk az $(i_1-1, i_1), (i_1, i_1+1)$ és (j_1, j_1+1) éleket, majd a második részlépésben beillesztjük az $(i_1-1, i_1+1), (j_1, i_1)$ és (i_1, j_1+1) éleket. A harmadik részlépés eltávolítja a $(j_2-1, j_2), (j_2, j_2+1)$ és (i_2, i_2+1) éleket, a negyedik pedig beilleszti a $(j_2-1, j_2+1), (i_2, j_2)$ és (j_2, i_2+1) éleket. Az α^* jelöléssel ezt a lépést így írhatjuk: $(1,1)$.

A kicseréléshez szükséges, hogy $\{i_1-1, i_1, i_1+1, j_1, j_1+1, i_2, i_2+1, j_2-1, j_2, j_2+1\}$ csúcsok legyenek páronként diszjunktak, és az i_1, j_1 és i_2, j_2 csúcsokat kicserélő lépések legyenek költség- és sorrendfüggetlenek.

Számos alkalmazásban a kicserélést úgy alkalmazzuk, hogy a 10 csúcs helyett két 3-3 pontból álló útrészlet két középső csúcsát cseréljük ki. Azaz ha a csúcsokat $\{i-1, i, i+1, j-1, j, j+1\}$-gyel jelöljük ahol az i_k-k és j_ℓ-ek alkotnak egy-egy utat. A lépés során eltávolítjuk $(i-1, i), (i, i+1), (j-1, j), (j, j+1)$ éleket, majd beillesztjük az $(i-1, j), (j, i+1), (j-1, i), (i, j+1)$ éleket. Ezt a speciális kicserélést felcserélésnek hívjuk.

Az $(1,1)$-kicseréléssel kapott szomszédság mérete $\mathcal{O}(n^4)$, amit le lehet csökkenteni

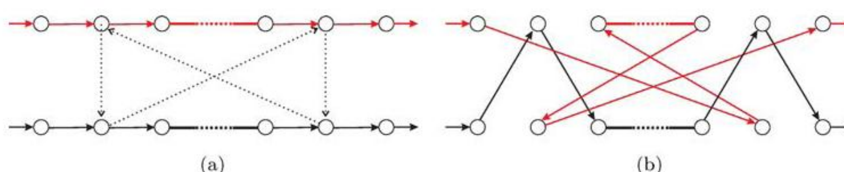
$\mathcal{O}(n^2)$-re, ha kicserélés helyett felcserélést használunk.

λ-felcserélés

Ez a szomszédság az előző általánosítása, mivel 1 helyett λ csúcsot helyezünk át egyik szegmensből a másikba. Emellett elvárjuk, hogy a törölt csúcsok helyére illesszük be az áthelyezendő csúcsokat.

A λ-felcserélés szomszédságnál, a törlendő $2k$ csúcsot ($k \leq \lambda$), $\binom{n}{2k}$-féleképpen választunk ki. Ezután $\mathcal{O}(n^{2k})$, ($k < n$) lehetőség van visszailleszteni a csúcsokat. Így a szomszédság mérete $\mathcal{O}(\sum_{k=0}^{\lambda} \binom{n}{2k} n^{2k}) = \mathcal{O}(\binom{n}{\lambda} n^{2\lambda}) = \mathcal{O}(n^{4\lambda})$.

Azonban nem minden λ-felcserélést lehet visszavezetni kicserélésre, mivel ahhoz a kicserélésben résztvevő csúcsoknak függetleneknek kell lenniük. Egy ilyen ellenpéldát láthatunk a következőképen.



2.1. ábra. a) A lépés előtt (a pontozott vonalak az eltávolítandó éleket jelölik), b) A lépés utáni állapot

2.3.2. Él-cserélő szomszédságok

Ezt a szomszédságot akkor használjuk, ha a közvetlenül érintett élek száma nagyobb, mint a csúcsoké. Az útvonaltervező és ütemező feladat heurisztikájában a legtöbbször ezt használják.

Általános leírás

Ha kivesszük a $d_1, d_2, ..., d_k \in A$ különböző éleket az óriástúrából, akkor az k db rész útra esik szét$(s_1, s_2, ..., s_k)$. Ezeket a következőkben szegmenseknek fogjuk hívni. Ahhoz, hogy megint óriástúrát kapjuk, hozzá kell venni k db élt a szegmensekhez. A hozzáadott éleket jelölje $a_1, a_2, ..., a_k \in A$. $\forall a_i$-re annak kell teljesülnie, hogy az él legalább egyik csúcsa kapcsolódik valamely szegmens kezdő vagy végpontjához. Azaz minden csúcsra azonos számú törölt és hozzáadott él illeszkedik. Ezért a $G = (d_1, ..., d_k, a_1, ..., a_k)$ feszítőgráfot

fel lehet bontani a törölt és beillesztett élekből álló alternáló körökre.

Attól függően, hogy milyen $a_1, a_2, ..., a_k$ éleket illesztünk be, a szegmensek permutálódhatnak, invertálódhatnak. Ez utóbbi azt jelenti, hogy a szegmens irányítása megváltozik, azaz fordított sorrendben megyünk végig a pontokon.
Így minden élcserélő m lépést az alábbi négy műveletre bonthatunk szét:

1. k-**szegmentálás**. Ez eltávolít k db élt az óriástúrából, így megkapjuk az $s_1, s_2, ..., s_k$ szegmenseket.

2. k-**inverzió**. Ez invertálja a k db szegmens egy részhalmazát, azaz $\forall i : s_i^{\pm 1}$-t képezzük.

3. k-**permutáció**. Ez megváltoztatja a szegmensek sorrendjét, így kapva $s_{\pi(1)}^{\pm 1}, s_{\pi(2)}^{\pm 1}, ..., s_{\pi(k)}^{\pm 1}$-ket.

4. k-**összefűzés**. Ez az eljárás összefűzi a szegmenseket, ami egy új, x' óriástúrát eredményez.

Ha minden beillesztett él különbözik a törölt élektől, akkor a lépést valódi lépésnek hívjuk.

Minden permutáció felbontható diszjunkt ciklusok szorzatára. (Ez a felbontás a ciklusok sorrendjétől eltekintve egyértelmű.)

π alkalmazása után $s_{\pi(1)}^{\pm 1}, s_{\pi(2)}^{\pm 1}, ..., s_{\pi(k)}^{\pm 1}$-et akkor és csak akkor lehet összefűzni, ha π ciklikus permutáció . A nem ciklikus permutációk többszörös köröket eredményeznek, amelyek nem feltétlenül vannak összhangban egy megengedett útvonaltervvel.

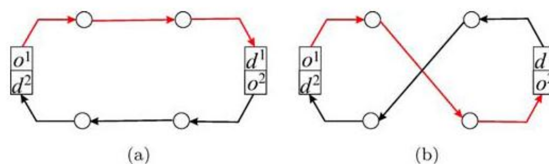
A 2. és 3. művelet meghatározza a lépés típusát. Ahhoz, hogy le tudjuk írni a különböző típusokat, bevezetjük az abc-jelölést.

a szegmensek jelölésére az ABC első k db kisbetűjét használjuk. Nagy betűvel jelöljük, ha egy szegmens invertálva került bele az óriástúrába. A "|" jel a körök elválasztására szolgál. Például: $aBDc$ - a lépés után a 2. és 4. szegmens invertált, a 3. és 4. pedig helyet cserélt. Másik példa: $aC|bd$ - az 1. és az invertált 3. szegmens alkot egy kört, illetve a 2. és a 4. alkot egy másikat. Így azonban a jelölés függ az óriástúrán belül a szegmensek betűzésétől, ezért a következőképpen egyértelműsítjük: tegyük fel, hogy az a szegmens rögzítve van egy megfelelő definícióval, pl.: a tartalmazza az 1-es indexű csúcsot. Így minden ciklikus permutáció egyedien írható le úgy, hogy az első betűje a. Tehát (a, c, b, d)-t nem lehet úgy írni például, hogy (d, b, a, c). Ebből adódóan $(k-1)!$ permutációt különböztetünk meg. A több körből álló esetben az első kör a-val kezdődik, a második kör pedig a maradékból szegmens közül a legelső betűvel, stb. Pl.: $aDf|c|eB$ helyett $aDf|Be|c$-t kell írni.

Szimmetrikus esetben x' és x'^{-1} (inverz megoldás) óriás túrák azonos megoldást adnak.

Pl.: Ab ugyanazt az óriástúrát adja, mint aB. Ezzel ellentétben az aszimmetrikus esetben számításba kell venni az első szegmens irányítását is.

A kezdő és végpontok és az inverz szegmensek hozzárendelése. Minden útnak van egy kezdő és egy végpontja. Bármely lépés hatással lehet legalább 2 útra úgy, hogy elrontja a kezdő és végpontok sorrendjét. Például a lépés hatására az egyik út kezdőpontból kezdőpontba, a másik pedig végpontból végpontba megy, ahogy a következő képen láthatjuk:



2.2. ábra. a) A csere előtt, b) A csere után

Ha a kezdő és végpontokat a lépés után helyre szeretnénk állítani, akkor egy optimalizálási feladatot kell megoldanunk. Ahol még 2 út esetén is 4 lehetőség van (ha a kezdő és végpontok kompatibilisek).

Azért, hogy ne kelljen közben megoldanunk további optimalizálási problémákat, a szegmensek invertálását egy egyszerű módon implementáljuk:

1. Ha s kizárólag követelményt támasztó csúcsokból áll, akkor közvetlenül invertálható.

2. Ha s tartalmaz egy (o, d) párt: $s = (v_1, v_2, ..., v_p, t, s, w_1 w_2, ..., w_q)$, akkor $s^{-1} = (w_q, ..., w_2, w_1, d, o, v_p, ..., v_2, v_1)$. Az inverzzel megengedett megoldást kapunk, mivel a 2. és 3. művelet után az óriástúra még mindig o és d kezdő és végpontból és néhány követelményt támasztó pontból áll, és a kezdő és végpont megfelelő sorrendben követi egymást.

3. Ha s több különböző utat tartalmaz: $r^{h(i)}, r^{h(i)+1}, ..., r^{h(j)}$, azaz $s = (v_1, v_2, ..., v_p, d, r^{h(i)}, r^{h(i)+1}, ..., r^{h(j)}, o, w_1 w_2, ..., w_q)$. Ekkor az $s^{-1} = (w_q, ..., w_2, w_1, d, r^{h(i)}, r^{h(i)+1}, ..., r^{h(j)}, o, v_p, ..., v_2, v_1)$. Ami azt jelenti, hogy a közbe eső utakat nem kell invertálni. Ekkor az előző esethez hasonlóan megengedett megoldást kapunk, mivel a megengedettséghez elég, ha a közbe eső kezdő és végpontok megfelelő sorrendben vannak, ez pedig teljesül.

k-Opt

Ez a legrégebbi és a legszélesebb körben használt élcserélő szomszédsági típus. Ennek során $k \geq 2$ élt eltávolítunk az óriástúrából, majd ugyanannyi élt beillesztünk, úgy hogy a π permutáció ciklikus legyen. Habár a legtöbbször egy út megtalálására alkalmazzák, általánosítható többszörös út megtalálására is.

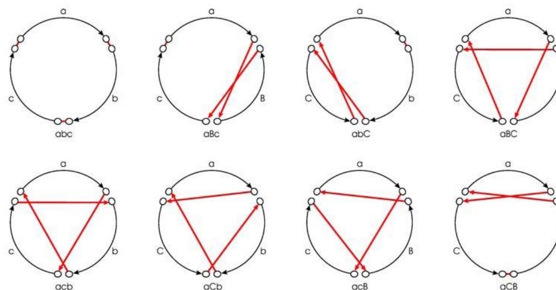
Ha a beillesztett és eltávolított élek páronként nem különbözők, akkor nem valódi lépést kapunk, ezért ezt k'-Opt lépésként is tekinthetjük ($k' \leq k$).

$\forall k \geq 2$-re a k-Opt szomszédság méretét kiszámolhatjuk az előző definíció alapján.

- $\binom{n}{k}$ lehetőség van kiválasztani k db élt
- 2^{k-1} lehetőség van végrehajtani inverziót
- $(k-1)!$ lehetőség van elvégezni ciklikus permutációt

Így a szomszédság mérete: $\binom{n}{k}2^{k-1}(k-1)!$. Ez felső becslés, mivel a szegmensekre bontás nem invertálható szegmenseket is eredményezhet.

Adott k-ra k-inverzió és k-permutáció végrehajtása után $2^{k-1}(k-1)!$-féle lehetséges szegmenset kaphatunk. Ezt a számot jelöljük MT-vel. Például $k = 2$-re és $k = 3$-ra $MT(2) = 2$, $MT(3) = 8$. Azonban a szegmensekre bontás során kaphattunk azonos felbontást is, különböző k-inverzió és k-permutáció végrehajtásával. Például a két darab 2-opt lépés azonos, mivel mindkettő az $(i, i+1)$ és $(j, j+1)$ éleket $(i, j\ )$ és $(i+1, j+1)$ élekkel helyettesíti, és az $i+1$. és j. csúcsok közötti szegmenst invertálja. A nyolc darab 3-opt lépés a következő:



2.3. ábra. A 8 darab 3-opt lépés

Jelöljük $PMT(k)$-val a k-opt lépések közül a valódi lépések számát. Ekkor $PMT(2) = 1$ és $PMT(3) = 4$. Általánosan a $PMT(k)$ értékét a következő képpen számolhatjuk. Miután

kiválasztottuk, melyik k élt szeretnénk törölni az óriástúrából, az $MT(k)$-ból ki kell vonni azon lépések számát, melyek kevesebb, mint k éllel generálhatóak. Az utóbbiak pontosan a $PTM(i)$ lépések, ahol $i \leq k$. Nem valódi lépés továbbá az identitás, amely pontosan a törölt éleket illeszti vissza. Ebből adódik a rekurzív formula: $PMT(k) = MT(k) - \sum_{i=2}^{k-1} PMT(i) - 1$, ha $k \geq 3$ és $PMT(2) = 1$.

A valódi lépések számának és az összes lépés számának aránya a kis esetekben ($k = 100$-ig) $0,5$ és $0,6$ között van, azonban meg kell jegyezni, hogy ez az arány monoton nő, ha $k \to \infty$.

Lin-Kernighan

Ahogy az általános leírásban bevezettük, a törölt éleket $d_1, \ldots, d_k$-val, a hozzáadott éleket $a_1, \ldots, a_k$-val jelöljük. Ezek az élek alternáló köröket alkotnak. Azokat a k-opt lépéseket, ahol $d_1, \ldots, d_k, a_1, \ldots, a_k$ élek egy alternáló kört alkotnak, SAC-k-opt lépésnek nevezzük. A 2-opt és 3-opt lépések azonosak SAC-2-opt és SAC-3-opttal, de ha $k \geq 4$, akkor a SAC-k-opt valódi részhalmaza k-optnak. Például *acbd* 4-opt, de nem SAC-4-opt.

Címkézzük meg az alternáló kör $2k$ csúcsát: $i_1, \ldots, i_k, j_1, \ldots, j_k$. Ezek a csúcsok determinálják a lépést, mivel a törölt és beillesztett éleknek így kell egymást követniük: $d_p = (i_p, j_p)$, $a_p = (j_p, i_{p+1})$, $p = 1, \ldots, k$ (feltettük, hogy $i_{k+1} = i_1$). Ahhoz, hogy a lépés a Lin-Kernighan családba tartozzon, az a_i, d_j csúcsok mindegyikének páronként diszjunktnak kell lennie.

A Lin-Kernighan lépés mindig egy él törlésével kezdődik, amelyet ha törlünk, akkor egy Hamilton-utat kell kapjunk. Ez szükséges feltétel arra, hogy Lin-Kernighan lépést kapjunk.

Mivel a törölt és beillesztett élek páronként diszjunktak, a Lin-Kernighan lépés pontosan k élt cserél fel másik pontosan k db éllel. Tehát a Lin-Kernighan lépések a valódi SAC-k-opt lépések valódi részhalmaza.

2.4. Keresési technikák

Ebben a fejezetben a keresési technikákról, és azok megfelelő megválasztásáról lesz szó. Továbbra is feltesszük, hogy x egy adott megoldása az útvonal tervezési problémának, és szeretnénk megtalálni a megoldások között a lokális optimumot. A keresési technikákat két fő csoportra oszthatjuk: közvetlen és indirekt keresési technikára. A közvetlen keresési

technika felsoroláson alapszik, és így direkt módon találja meg a javító megoldást. Az indirekt keresési technika átalakítja a javító szomszéd keresését optimalizálási feladattá, például a legrövidebb út keresésére.

A megfelelő keresési technika megválasztásához ismernünk kell a szomszédság típusát, a költségfüggvényt, és a korlátokat, mivel ezek döntik el, hogy létezik-e költségfüggetlen dekompozíciója a lépéseknek. Ha a költség nem függ az élektől és a járművektől, akkor csak a szomszédság határozza meg, hogy a lépést költségfüggetlenül lehet-e részlépésekre bontani. Ha viszont a költség függ az erőforrás változóktól, akkor nem létezik költségfüggetlen felbontás.

Figyelembe véve a korlátokat, kétféle technikával garantálhatjuk a szomszéd megoldás megengedettségét. Vagy a keresés alatt végig ellenőrizzük a korlátokat, vagy részlépések kompozícióját használjuk fel a szomszédok keresésekor, ami szintén garantálja a megengedettséget.

2.4.1. Közvetlen keresés

Közvetlen keresésről azokban az esetekben beszélünk, ahol a szomszédságot csúcsok vagy élek cseréjével kapjuk. Tegyük fel, hogy a szomszédság mérete $\mathcal{O}(n^k)$, azaz a csúcsok számában polinomiális. Azért, hogy a keresés hatékonyságát növeljük, a keresés korai szakaszában kizárjuk a lehető legtöbb nem javító, vagy nem megengedett szomszédos megoldást.

A probléma a következő: k db élt (csúcsot) kell kicserélnünk másik k db élre (csúcsra) úgy, hogy javító szomszédos megoldást kapjunk. Minden közvetlen keresés egyesével választja ki a k darab élt (csúcsot). Ezért az a cél, hogy ezt a lehető legkevesebb élkombináció (csúcskombináció) megvizsgálásával tegyük meg $\forall i : 1 \leq i \leq k$. Némely esetben be lehet bizonyítani, hogy nem, vagy csak kevés lehetséges él (csúcs) marad az i. él (csúcs) kiválasztása után. A két fő szempont, ami alapján ki lehet zárni éleket (csúcsokat), a költség és a megengedettség. A költség alapján kizárhatunk éleket (csúcsokat), ha be tudjuk látni, hogy nincsen olyan javító szomszéd, amely tartalmazná az eddig kiválasztott i db élt (csúcsot). A megengedettség alapján is hasonlóképpen zárhatjuk ki az éleket (csúcsokat), csak ebben az esetben azt kell bizonyítani, hogy nincsen olyan megengedett csere, amely tartalmazná a kiválasztott éleket (csúcsokat). Sajnos azoknak az eseteknek a száma, amikor mindkét kizárási módszert egyszerre használhatjuk nagyon kevés, ezért általában még a keresés elején döntenünk kell az egyik kizárási stratégia mellett.

A következőben két közvetlen keresési technikát fogunk bemutatni. A lexikografikus keresést, ami a megengedettség alapján zárja ki a további beválasztható éleket (csúcsokat). Illetve a szekvenciális keresést, amely a költség alapján csökkenti a vizsgálandó élek (csúcsok) számát.

Lexikografikus keresés

A lexikografikus keresésnél kihasználjuk, hogy az élek (csúcsok) az útvonaltervben elfoglalt helyük alapján sorszámozhatók. Az algoritmus alapjaként természetesen adódik, hogy a k élt (csúcsot) k db indexszel keressük meg, amelyek a következőképpen futják be az n db élt (csúcsot): $i_1 := 1, \ldots, n$; $i_2 := i_1+1, \ldots, n$; $\ldots$; $i_k := i_{k-1}, \ldots, n$. Az algoritmus pontos lépései azonban a szomszédság típusától függnek.

k-Opt. Tegyük fel, hogy adva van, hogy melyik k él lett törölve, ezáltal a szegmensek rögzítettek. Továbbá tegyük fel, hogy az algoritmus az i. él kiválasztásánál tart. Ekkor az algoritmus tovább léphet vagy leállhat anélkül, hogy mind a k db beillesztendő élt kiválasztaná, ha bizonyítani tudjuk, hogy az első i szegmens összeláncolása az eddig kiválasztott élekkel nem megengedett szomszédot eredményez. Például ahogy az él-cserélő szomszédságok általános leírásánál láthattuk, előfordulhat, hogy két szegmens összefűzésekor nem megengedett szomszédot kaptunk, mivel a két út kezdőpontból kezdőpontba, illetve végpontból végpontba ment.

Tehát a lexikografikus keresés legfőbb előnye abban rejlik, hogy a szegmensek szisztematikusan épülnek fel. Minden lépésben csak 1 élt (csúcsot) törlünk, vagy illesztünk be, ezért a korlátokat tudjuk konstans időben ellenőrizni.

Szekvenciális keresés

A szekvenciális keresést azokra a problémákra alkalmazzák, ahol k db élt (csúcsot) kell helyettesíteni másik k db éllel (csúccsal). Az alapötlete a keresésnek az, hogy a szomszédságot a részlépések alapján rekurzívan nézzük végig. Így a lexikografikus kereséssel ellentétben nem sorszám alapján haladunk, hanem a jelölteket költség szerint növekvő sorrendben listázzuk. Ez a keresés nagyon gyors, ha a lista méretét $\mathcal{O}(n)$-ről egy kis konstansra tudjuk csökkenteni. A hátránya viszont, hogy a megengedettséget csak a k. él

(csúcs) kiválasztása után tudjuk ellenőrizni legjobb esetben is lineáris időben. Az elképzelhető legrosszabb forgatókönyv az, ha sok javító, de nem megengedett megoldást kell ellenőriznünk.

Ahhoz, hogy ezt a keresési módszert használhassuk, két feltételnek kell teljesülnie: a szomszédságnak ciklusfüggetlennek és költségfüggetlennek kell lennie. Míg az előbbi feltétel elmaradása esetén még lehet az algoritmust módosítani (Lin-Kernighan heurisztika, amit ebben a fejezetben fogunk tárgyalni), a második feltétel szükséges és elégséges. Ezért a továbbiakban feltesszük, hogy a szomszédságot generáló lépés költségfüggetlenül bontható részlépésekre.

A keresés előnyét a következő tétel mondja ki, melyet Lin és Kernighan igazolt 1973-ban.

1. Tétel: *Ha számok egy sorozatának* $(g_i)_{i=1}^k$ *az összege pozitív* $\sum_{i=1}^k g_i > 0$*, akkor létezik olyan ciklikus permutációja* π *ezen számoknak, hogy minden részletösszeg is pozitív* $\forall p$: $1 \leq p \leq k \sum_{i=1}^p g_{\pi(i)} > 0$.

2-*Opt.* A 2-opt lépések egyben SAC-2-opt lépések is, ebből adódóan ciklikusak. A 2-opt szomszédságot generáló lépéseket két részlépésre bontjuk. Az első részlépés legyen $p_1 := p^{\text{add}}_{j_1,i_2} \circ p^{\text{del}}_{i_1,j_1}$, ami törli a $d_1 = (i_1, j_1)$ élt és beilleszti az $a_1 = (j_1, i_2)$ élt. A második részlépés: $p_2 := p^{\text{add}}_{j_2,i_1} \circ p^{\text{del}}_{i_2,j_2}$, ami hasonlóan az előzőhöz törli a $d_2 = (i_2, j_2)$ élt, és beilleszti az $a_2 = (j_2, i_1)$ élt. Az élek költségét jelölje $|d_\ell|, |a_\ell|, \ell = 1, 2$. Ekkor a részlépések haszna a következőképpen számolható: $g_1 = |d_1| - |a_1|, g_2 = |d_2| - |a_2|$. Ahhoz, hogy javító szomszédot kapjunk, teljesülnie kell a hasznossági kritériumnak: mindkét haszonnak g_1-nek és g_2-nek is nagyobbnak kell lennie 0-nál.

Javító 2-opt lépést a következőképpen keresünk szekvenciális kereséssel. Fusson $i_1 = 1 \dots n$ $(n = |V|)$, ehhez keresünk j_1 csúcsot i_1 szomszédai között. Ezzel megkaptuk d_1 élet. Most olyan a_1 élt kell keresni, amire $|a_1| < |d_1|$. Ilyen élt általában gyorsan találunk, ha listázzuk a j_1-hez kapcsolódó éleket költségük szerint növekvő sorrendben. Ha találtunk egy ilyen élet, akkor már csak egyféleképpen fejezhetjük be a lépést, ha megengedett szomszédos megoldást szeretnénk kapni. Miután elvégeztük ezt a javító lépést, innentől ezt tekintjük megoldásnak és ezt javítjuk egy másik d_1 él törlésével kezdődő cserével.

Lin-Kernighan. A Lin-Kernighan lépés annyiban különbözik a 2-opt lépéstől, hogy pontosan k db él töröl és illeszt be az éppen javítandó x megoldásba, ahol a törölt és hozzáadott élek egy alternáló kört alkotnak. Tegyük fel, hogy az első i db élt töröltük, és helyettük már beillesztettünk i db élt, azaz $d_1, \dots, d_i$ és $a_1, \dots, a_i$ adott. Minden d_i-t

törlő és a_i-t hozzáadó részlépés haszna $g_i = |d_i| - |a_i|$. Ez után a d_{i+1}. törlendő élt úgy kell kiválasztani, hogy illeszkedjen a_i végpontjához, és a törlésével Hamilton-utat kapjunk. Az a_{i+1} élnek a következő kritériumot kell teljesítenie: $|a_{i+1}| \leq G_i - |d_{i+1}|$. Ha ilyen él nincsen, akkor az algoritmus ennél a pontnál leállhat. Ebből a feltételből következik, hogy a keresés mélységét dinamikusan kell tárolni. Ezért a Lin-Kernighan algoritmus a mélységi bejárás elvén működik azzal a kiegészítéssel, hogy minden iterációban a hozzáadott élt a hasznossági kritérium szerint választjuk ki.

Felcserélés. Ahogy az előző fejezetben leírtuk, a felcserélés az i és j csúcsot cseréli fel az $(i-1,i), (i,i+1), (j-1,j), (j,j+1)$ élek törlésével és $(i-1,j), (j,i+1), (j-1,i), (i,j+1)$ élek beillesztésével. Ezt a lépést bontsuk fel két részlépésre a következőképpen. Legyen p_{ij} az a részlépés, amelyik törli a $(i-1,i), (i,i+1)$ éleket, és beilleszti a $(j-1,i), (i,j+1)$ éleket. Ekkor $m_{ij}^{\text{swap}} = p_{ij} \circ p_{ji} = p_{ji} \circ p_{ij}$. A részlépés haszna: $g(p_{ij}, x) = c(i-1,i) + c(i,i+1) - c(j-1,i) - c(i,j+1)$. Ahhoz, hogy javító szomszédot találjunk, ennek a haszonnak pozitívnak kell lennie.

A keresés menete a következő. Először kiválasztjuk i-t, és keressük hozzá j-t, amire teljesül a hasznossági kritérium. Ezt egyszerűen megtehetjük, ha i szomszédait költségük szerint növekvő sorrendben listázzuk. Legyen $\alpha = \frac{c(i-1,i)-c(i,i+1)}{2}$, ami fix, ha i rögzített. Ezután a hasznossági kritériumot át lehet értelmezni: $g(p_{ij}, x) > 0 \Leftrightarrow (c(j-1,i) - \alpha) + (c(i,j+i) - \alpha) < 0 \Leftrightarrow c(j-1,i) < \alpha \vee c(i,j+1) < \alpha$. Tehát elég a törölt és beillesztett élek költségét összehasonlítani α-val, ami konstans időben ellenőrizhető.

Összegzés. A szekvenciális keresés nagyon hatékony, ha költség alapon szeretnénk keresni, azonban a jelöltlista összeállításánál fel kell tenni, hogy a részlépések költségfüggetlenek. Ezért ezt a keresést akkor lehet alkalmazni, ha például kilométert szeretnénk minimalizálni. Ha az útvonalak időtartamát, vagy a visszaérkezés időpontját szeretnénk minimalizálni, arra ez a keresési technika nem alkalmas.

2.4.2. Optimalizálás alapú, indirekt keresési technikák

Az optimalizálás alapú keresési technika ötlete az, hogy a problémát átírjuk a legjobb lépés megtalálására az adott szomszédságban. Ennek több megközelítése is lehetséges, de mi csak a lépés-kompozíción alapuló változatot tárgyaljuk. Ez a megközelítés explicit térképezi fel az aktuális megoldásra alkalmazható részlépéseket, és alakítja át a döntést egy

segédproblémává.

A megközelítés az úgynevezett nagyméretű szomszédsági keresést használja. Ennek az algoritmusa általánosságban így írható le:

1. Az algoritmus inputként kap egy $x^0 \in X$ kezdeti megoldást, továbbá az iteráció számlálóját beállítjuk nullára, azaz $t := 0$

2. Ismételjük az alábbi ciklust:

3. Generáljuk le az aktuális megoldás szomszédságához tartozó optimalizálási problémát, $P(x^t)$-t.

4. Keressük az optimális-, de legalábbis javító megoldást $P(x^t)$-ben.

5. Ha létezik ilyen x', akkor $x^{t+1} := x'$ és $t := t + 1$

6. Leállunk, ha nincs javító szomszéd, mert ez azt jelenti, hogy lokális optimumot kaptunk.

Az algoritmus időigényes része az optimalizálási feladat generálása, és megoldása. Mivel az optimalizálási feladat $\mathcal{NP}$-nehéz is lehet, fontos a 4. lépésben a megfelelő heurisztika alkalmazása.

Lépés-kompozíció alapú megközelítés

A megközelítés alapötlete, hogy a nagyméretű szomszédságot, bontsuk fel a részlépések halmazára. Ezen részlépések által meghatározott optimalizálási probléma eredménye a lépések egy részhalmaza, amelyek kompozíciója megengedett és maximális hasznú a szomszédságban. Az optimalizálási segédprobléma alapja egy gráf, amelyet javító gráfnak hívunk.

Körkörös javító gráf. A körkörös javító gráf azt modellezi, ahogyan egy megoldásból csúcsokat dobunk ki és helyettesítünk más csúcsokkal. A javító gráf csúcsai legyenek azok a csúcsok az eredeti gráfból, amelyek benne lehetnek egy úgynevezett csúcs-kidobó láncban. A csúcs-kidobó lánc hasonló a felcseréléshez, a különbség abban rejlik, hogy j-t nem i helyére teszi be, hanem egy harmadik csúcs helyére, amelyet egy negyedik helyére illeszt be, és így tovább. Végük a k. lépésben a k. csúcsot beilleszti az első, azaz i helyére.

A javító gráf valamely két csúcsát jelölje S_i és S_j. Ekkor az (S_i, S_j) irányított él annak a részlépésnek felel meg, amelyik eltávolítja S_j-t az aktuális pozíciójából, és S_i-t illeszti be a megüresedett helyre. Az (S_i, S_j) él költsége legyen a haszon ellentettje. Ekkor egy javító körkörös cseréhez elég találni egy negatív kört a javító gráfban. Azonban ez a feladat $\mathcal{NP}$-

nehéz, mivel megfeleltethető a legrövidebb út keresési problémának. Mivel a csúcs-kidobó láncok szomszédsága ciklikus, alkalmazhatjuk a hasznossági kritériumot (1. Tétel), ami ebben az esetben azt jelenti, hogy bármely negatív összköltségű körben minden részösszeg negatív kell legyen.

Teljes lépések kompozíciója. Ezt a módszert a nagyméretű szomszédságokban való keresésre használjuk. Működésének lényegét a 2-opt szomszédságon mutatjuk be. Legyen m megengedett, költségfüggetlen kombinációja 2-opt lépéseknek: $m = m^{2-\text{opt}}_{i_1,j_1} \circ m^{2-\text{opt}}_{i_2,j_2} \circ \cdots \circ m^{2-\text{opt}}_{i_t,j_t}$. Ekkor m-et összetett lépésnek hívjuk. Némely esetben ennek az összetett lépésnek a haszna nagyobb, mint a legjobb javító 2-opt lépés ($m^{2-\text{opt}}_{i_\text{best},j_\text{best}}$). Azonban a megengedettség vagy a jogosultsági kritériumok megakadályozhatják, hogy további 2-opt lépést hajtsunk végre, miután a legjobb 2-opt lépést végrehajtottuk.

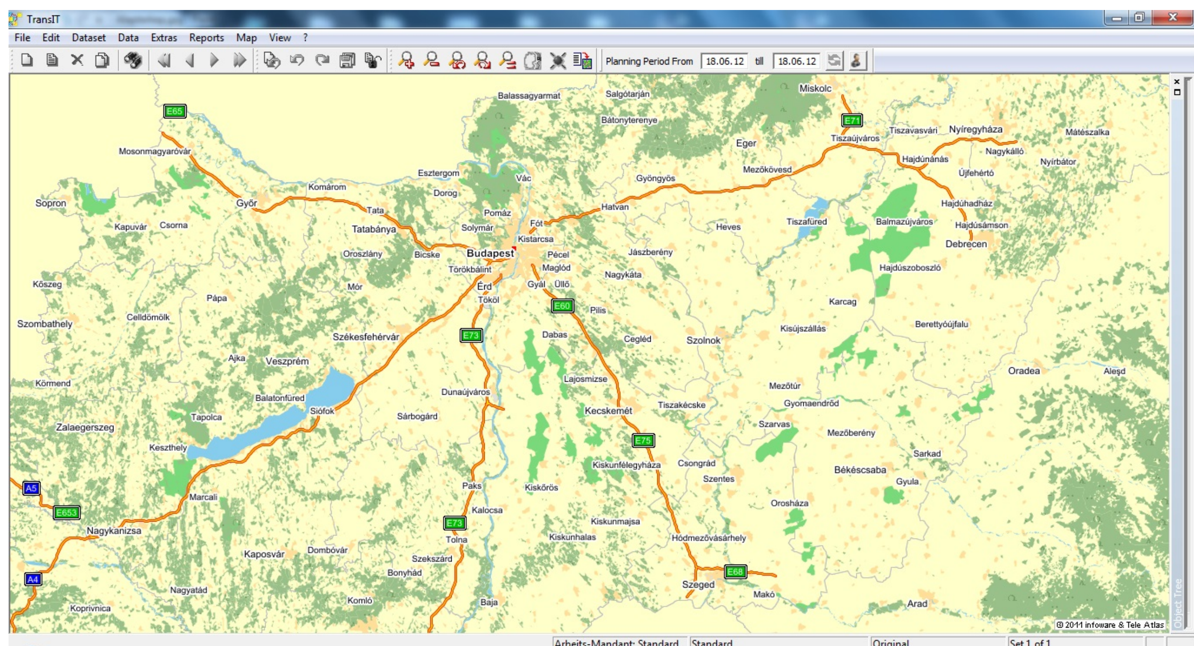
A 2-opt szomszédsághoz a következő javító gráfot rendeljük. Számozzuk újra a csúcsokat az aktuális megoldásban $1, \ldots, n$-ig. Egy 2-opt lépés leírható a javító gráf 1 élével: $(i, j) \in \{1, \ldots, n\}, i + 2 \leq j$. Ez felel meg a $d_1 = (i, i + 1), d_2 = (j - 1, j)$ élek törlésének, $a_1 = (i, j - 1), a_2 = (i + 1, j)$ élek beillesztésének és az $(i + 1, j - 1)$ szegmens megfordításának. Jelölje $G_2(i, j)$ az előbbi 2-opt lépés hasznát. Ekkor a javító gráf csúcsai legyenek $V := \{1, \ldots, n + 1\}$, ahol 1 és $n + 1$ ugyanannak a csúcsnak felel meg. Az irányított élet két csúcs közé akkor húzunk, ha az megengedett 2-opt lépés, azaz $A = \{(i, j) : \text{ahol } 2\text{−opt megengedett}\}$, továbbá $(i, i + 1)$ élt is behúzzuk, ha az élt az összetett lépés változatlanul hagyta. Az (i, j) él költsége $-G_2(i, j)$, ha $j \neq i + 1$, és 0, ha $j = i + 1$. Ekkor a legrövidebb út megtalálása 1-ből $n + 1$-be, megfelel a maximális költségcsökkentő összetett 2-opt lépésnek. Ha a legrövidebb út értéke negatív, akkor javító összetett megoldást találtunk, ha nem, akkor az aktuális 2-opt lépés optimális.

3. fejezet

A probléma megoldása a TransIT nevű szoftverrel

A TransIT egy útvonal optimalizálási szoftver, amelyet a GTS Systems and Consulting GmbH működtet. Beágyazott kereső algoritmusával, mely az előző fejezetekben leírtak alapján működik, találja meg a kezelési felületén beállított feltételeknek megfelelő túraútvonalakat. A szoftver mögött van egy adatbázis, ami tárolja a kezelőfelületen megadott adatokat. Természetesen közvetlenül az adatbázisba is írhatjuk az adatokat, a kezelőfelület a felhasználói élmény fokozására szolgál.

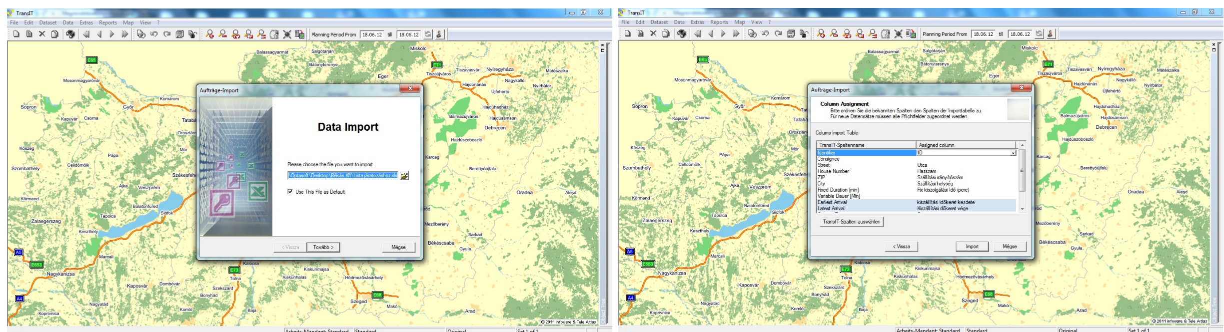
A szoftver megnyitásakor a következő kép fogad:



A problémában az alábbi paramétereket kellett megadni ahhoz, hogy elérjük a kívánt eredményt.

1. Megrendelések

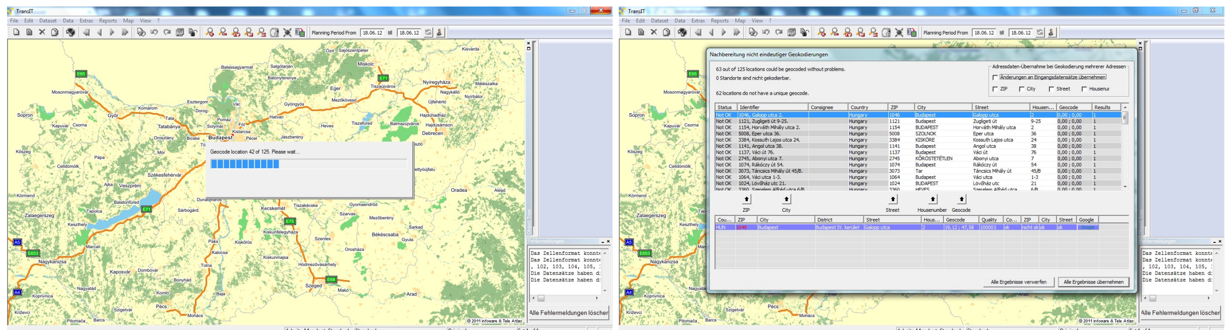
   A boltok adatait (földrajzi hely, áruátvétel ideje, boltban töltendő idő, megrendelés mennyisége) egy Excel táblában kaptuk meg a sütőipari termékeket szállító cégtől. Ebből importáltuk be az adatokat a TransIT adatbázisába.



   A bal oldali képen látható az Import Vehicle menüre kattintás hatására megjelenő felugróablak, ahol ki lehet választani az Excel táblát, amelyikből adatokat szeretnénk beolvasni. Ha ezt kiválasztottuk, utána be kell jelölni, hogy a táblázat melyik munkalapjáról szeretnénk beolvasni az adatokat. A jobb oldali ábra mutatja a következő lépést, ahol azt kell megadni, hogy a táblázat melyik oszlopa melyik adatnak felel meg a TransIT-ban.

   Fontos lépés, hogy miután beolvastuk a megrendeléseket, geokódolni kell őket. Ezt a TransIT 90%-ban automatikusan végzi el, csak ott kell az általa felkínált listából választani, ha több hasonló címet is talál. Ha egyáltalán nem találja meg a címet, kézzel arrébb lehet húzni a boltot a pontos helyére. A geokódolás pontossága nagy mértékben függ az általunk be vitt adatok milyenségétől. Ez azt jelenti, hogy ha például nem adunk meg házszámot, vagy rosszul adjuk meg, akkor kisebb valószínűséggel fogja kitalálni magától a szoftver, hogy mi pontosan melyik földrajzi helyre gondoltunk.

   A bal oldali ábrán látható, ahogy éppen fut a geokódolás. A jobb oldali ábrán látható a lista,ahol ki kell választani, hogy az ő általa felkínált lehetőségek közül melyiket választjuk.



A megrendeléseknél be kell állítani azt is, hogy milyen jellegű területen fekszik, azaz, hogy milyen behajtási engedéllyel rendelkező autó szállíthat oda árut.

2. Autó típusok

   A TransIT felületén létrehoztunk autó típusokat, hogy a teherautók bizonyos paramétereit a csoportosításuk alapján adhassuk meg. Itt állítottuk be, hogy melyik autónak milyen a teherbírása, és hova van behajtási engedélye. Be kellett még állítani, hogy mennyi a fogyasztása kilométerenként, illetve hogy milyen egyéb költségekkel kell számolni, ha ezzel a járművel fuvarozunk. Például az autópálya matricával rendelkező teherautóknál itt lehet megadni, hogy mennyibe kerül az éves matrica.

   Fontos, hogy a szállítás során a törvényi előírásokat is betartsuk. Ilyen törvényi előírás például, hogy mennyi ideig lehet vezetni egy huzamban egy teherautót, és egy ilyen periódus után mennyi pihenőidőt kell tartani. Ennek a beállítására is az autó típusok megadásánál van lehetőség.

3. Teherautók

   A cég 21 teherautóval rendelkezik, ezeket Excel táblából olvastuk be, hasonlóképpen, mint a rendeléseket.

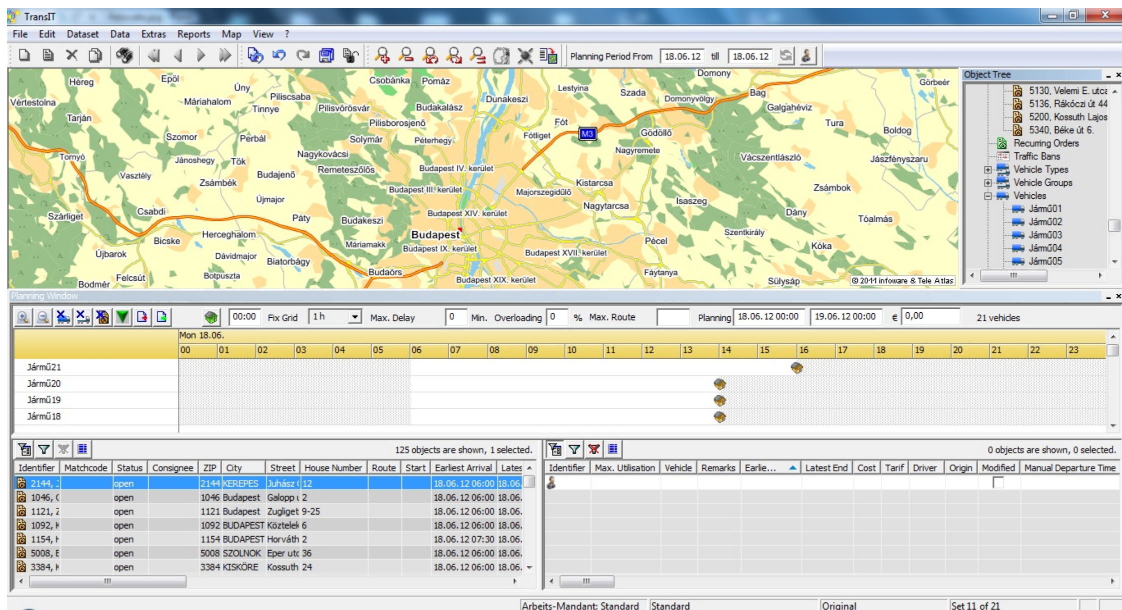
   A következőket kellett minden autónál megadni: a jármű nevét és azonosítóját, az alapértelmezett tartózkodási helyének címét, az időpontot, amikorra mindenképp a depóban kell lennie, illetve az intervallumot, amikor közlekedhet. Az utóbbi kettő elválhat egymástól, mert lehet hogy azt kérjük, hogy minden nap délben legyen a depóban egy ellenőrzésre vagy ebédre, de utána még mehetünk még egy kört. Megadtuk továbbá, hogy milyen típusú az autó, ezzel megadtuk az előző pontban felsorolt beállításait. Illetve, hogy milyen fajtájú megrendeléseket teljesíthet, amit a boltoknál úgy adtunk meg, hogy megmondtuk milyen behajtási engedéllyel rendel-

kező autó látogathatja meg.

4. Depó

   A feladatban egy depó szerepelt, így azt kézzel vittük fel a kezelő felületen keresztül. Itt meg kellett adni a depó pontos helyét, nyitvatartási idejét. Ebben a feladatban nem használtuk ki, de itt még be lehet állítani azt is, hogy mekkora átmenőforgalmat tud bonyolítani, illetve, hogy milyen autótípusokat tud befogadni. Ez utóbbi akkor lehet fontos, ha például valamiféle magasságkorlátozás van a depó területén, és az ennél magasabb típusú autók nem szállhatnak meg ott éjszakára.

Miután megadtuk az összes megrendelést, teherautót és feltételt, kiválaszthatjuk, hogy melyeket szeretnénk túraútvonalba betenni. Ehhez meg kell nyitnunk a tervező ablakot (Planning Window). Majd oda be kell húznunk azokat a megrendeléseket, és azokat a járműveket, amelyeket be szeretnénk tervezni az útvonalba. A mi esetünkben ez most az összes megrendelés és az összes jármű volt.

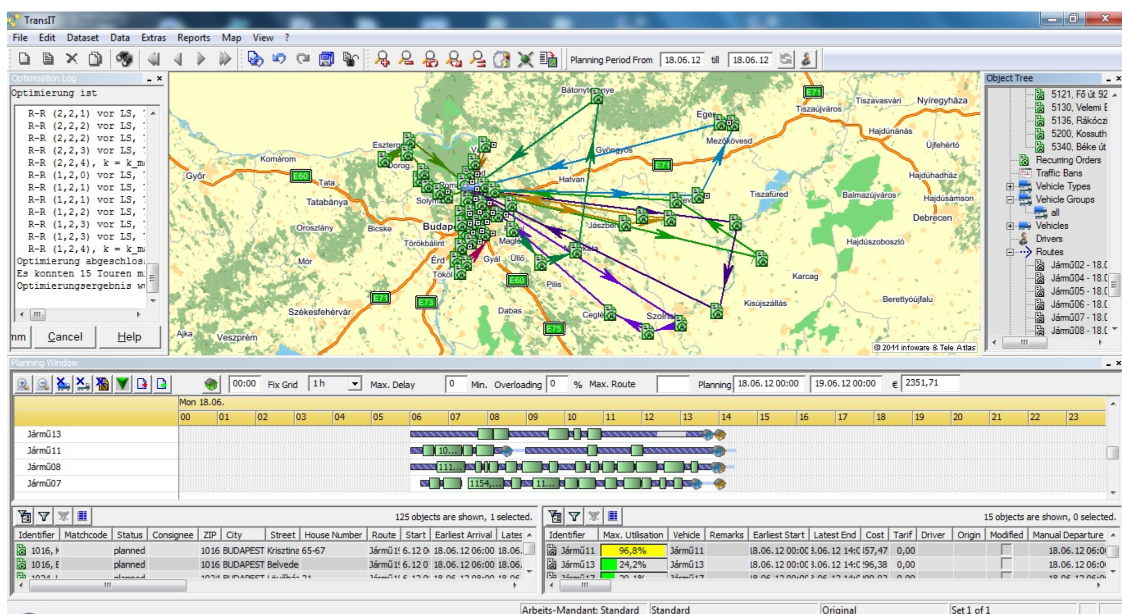


A kép alsó felében látható a tervező ablak, amely 3 részből áll: Járművek (fölső), Megrendelések (bal alsó), Útvonalak járműre lebontva(jobb alsó). Ez utóbbi az optimalizálás előtt még üres.

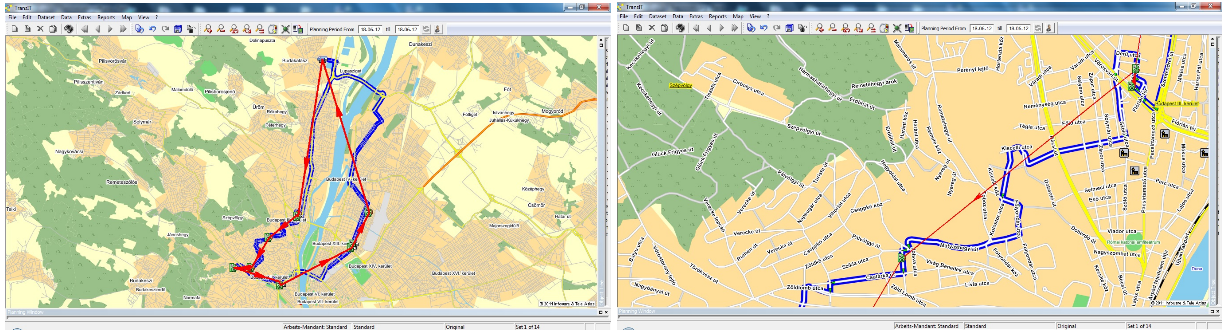
A tervező ablak menüsorában még további feltételeket adhatunk meg, illetve módosíthatunk a kapott eredményen. További feltételek: behúzhatunk egy plusz hazalátogatást a

depóba. Megadhatjuk a tervezési időszakot is, amire nagyon kell figyelni, hogy az az időszak legyen beállítva, amikorra tervezni szeretnénk, különben a tervezés hibára fog futni. Itt adhatjuk meg továbbá a maximális utak számát is.
Ha a tervezés lefutott, a kapott eredményen további módosításokat hajthatunk végre, például hozzáadhatunk további megrendeléseket a már elkészült útvonaltervhez. Erre van egy külön optimalizáló gomb, hogy ne kelljen megint az egész tervet lefuttatni, csak azt kiszámolni, hogy mely utakba tudjuk ezeket optimálisan betenni. Ez akkor hasznos, ha napközben jön egy S.O.S. megrendelés, és már mindenki úton van, így nem kell felülírni az egész aznapi tervet.

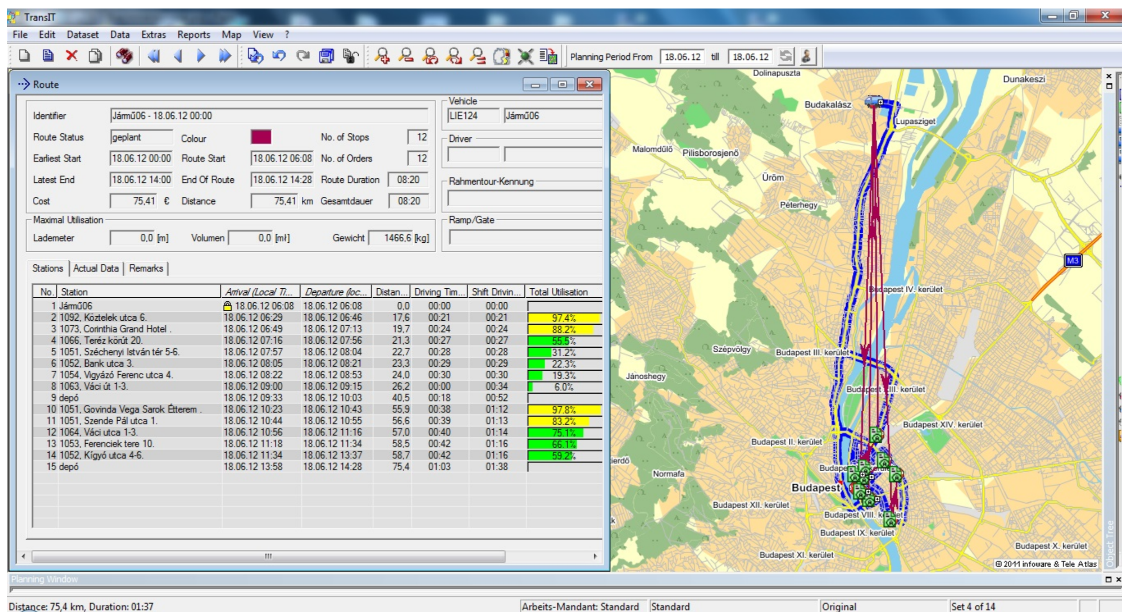
Most, hogy bevittük az összes korlátot, járműveket és megrendeléseket, elindíthatjuk a beépített optimalizálót. Ez az optimalizáló az előző fejezetben leírtak alapján működik, és keresi meg a lehető legjobb megoldást. Ha ez lefutott, a következő képen jelenik meg számunkra a képernyőn:



Az eddig barna, azaz még be nem tervezett megrendelések zöldre változtak, és nyilakkal vannak összekötve annak megfelelően, hogy milyen sorrendben látogatjuk meg őket. Minden járműhöz más színű nyíl tartozik. Természetesen a pontos útvonalat is meg tudjuk nézni, ahogy az a következő ábrákon látszik különböző felbontásban:



Nyomon követhetjük továbbá a jármű pontos útvonala mellett a leterheltségét, és hogy egy adott megrendeléshez mely időpontban érkezik. Ezt szemlélteti a következő ábra.



Most a kapott eredményeket fogjuk elemezni egy pár mondatban. Ahogy már korábban említettük, a cégnek 21 autó állt a rendelkezésére a feladat teljesítéséhez. Ebből a probléma megoldásakor 19-et használtak fel, melyekkel nagyságrendileg 3000 kilométert tettek meg. Az optimalizálással ezt az eredményt akartuk javítani.

Az optimalizálást két különböző beállítással végeztem el. Az egyik esetben maximálisan betartottam a napi 8 órás munkarendet, a másik esetben az autópálya matricával rendelkező autók munkaidejét egy órával meghosszabbítottam, mivel vidéki megrendelésből relatívan sok volt az autópálya matricás autókhoz képest. Így a következő eredményeket kaptuk:

Az első esetben 15 autóval sikerült teljesíteni az összes megrendelést. A járművek össze-

sen 2352 kilométert tettek meg, 47 óra alatt. Ezzel 21%-kal csökkentettük a teherautók számát, és 22%-kal az összesen megtett kilométerek számát.

A második beállítással, ahol a megnövelt munkaidő miatt a túlórákkal is számolnunk kell a következő eredményeket kaptuk. Az összes megrendelést 14 autóval sikerült teljesíteni, az autók összesen 2040 kilométert tettek meg, 43 óra alatt. Tehát ebben az esetben 26%-kal csökkentettük az autók számát, és 32%-kal az összesen megtett kilométerek számát. A 14 autó közül 3 volt autópálya matricás, így 3 túlórát kell fizetni ebben az esetben. Ennek ellenére az üzemanyag költség és az összes munkaóra száma miatt megéri ezt a beállítást választani.

Mindebből arra következtethetünk, hogy nagy piaci előnyre tehet szert az a cég, amelyik optimalizálási eszközzel tervezi meg erőforrásainak kihasználását.

Irodalomjegyzék

[1] BIRGER FUNKE, TORE GRÜNERT, *A Note on Single Alternating Cycle Neighborhoods for TSP*, Journal of Heuristics, 11: 135-146, 2005.

[2] BIRGER FUNKE, TORE GRÜNERT, *Local Search for Vehicle Routing and Scheduling Problems: Review and Conceptual Integration*, Journal of Heuristics, 11: 267-306, 2005.

[3] BIRGER FUNKE, TORE GRÜNERT, *TransIT User's Guide*, 2005.

[4] TransIT, *A TrasIT szoftver forráskódja*