Debreceni Egyetem

Informatikai Kar

Hatékony keresőalgoritmusok a mesterséges intelligenciában

Szakdolgozat

Témavezető
dr. Várterész Magda
egyetemi docens

Készítette
Legoza József
programtervező informatikus
hallgató

Debrecen
2009

Tartalomjegyzék

Bevezetés 4

I. Alapfogalmak 5

1. Állapottér-reprezentáció 5
1.1. Példa: Peg Solitaire 5
1.1.1. A probléma leírása 5
1.1.2. A probléma állapottér-reprezentációja 6

2. Gráfelméleti alapfogalmak 10

II. Az utazó ügynök problémája 11

3. A probléma állapottér-reprezentációja 12

4. Alapvető megoldási módszerek 13
4.1. A megoldás optimalitásának becslése 13
4.1.1. 1-fa korlát 13
4.1.2. Held-Karp korlát 13
4.2. Nyers erő (Brute force) 14
4.3. Heurisztikus módszerek 14
4.3.1. Útvonal-konstrukció 14
4.3.2. Útvonaljavítás 17

5. A Lin-Kernighan algoritmus 18
5.1. Az eredeti eljárás 19
5.1.1. Az algoritmus 20
5.2. Lin és Kernighan finomításai 24
5.3. A módosított algoritmus 25
5.3.1. Kiválasztott élek halmaza 25
5.3.2. Alapvető lépések 27
5.3.3. Kezdeti útvonalak 27

III. Lokális keresőalgoritmusok 28

6. Lokális keresési problémák 28
6.1. A szomszédsági gráf 29
6.2. A lokális keresés alapalgoritmusa 29

7. Szimulált hűtés 31
7.1. A módszer eredete, fizikai analógia . . . 31
7.2. Metropolis algoritmusa . . . 31
7.3. A szimulált hűtés algoritmusa, hűtési ütemtervek . . . 32
7.3.1. Megengedett megoldások előállítása . . . 34
7.3.2. Megoldás-kiértékelés . . . 35
7.3.3. Hűtési ütemtervek . . . 35
7.4. Előnyök, hátrányok . . . 36
7.5. Alkalmazási területek, változatok . . . 37
7.5.1. A módszer alkalmazása az utazó ügynök problémája esetén . . . 37

8. Tabu-keresés 37
8.1. A módszer alapötletei . . . 38
8.2. A memória effektív használata . . . 41
8.2.1. Változó hosszúságú tabu-lista . . . 41
8.2.2. Rövidtávú memória . . . 41
8.2.3. Középtávú memória . . . 42
8.2.4. Hosszútávú memória . . . 43
8.3. Algoritmus . . . 43
8.4. Alkalmazások . . . 44

Összefoglalás 45

Köszönetnyilvánítás 46

Irodalomjegyzék 47

Bevezetés

A mesterséges intelligencia hallatán valószínűleg sokak előtt a tudományos-fantasztikus irodalom termékei jelennek meg, azonban ez a tudományág jelenleg a számítástudomány fontos részét képezi. A szakirodalom intelligens ágensek tervezésének és vizsgálatának tudományaként írja le ezt a területet, ahol az intelligens ágens egy olyan rendszer, mely érzékeli környezetét, és olyan cselekedeteket hajt végre, mellyel maximalizálja saját sikerét.

Bár a mesterséges intelligencia célja kezdetben annak bizonyítása volt, hogy az emberi intelligencia gépek segítségével is szimulálható, mára egy igen szerteágazó tudománnyá nőtte ki magát: a gazdaság- és orvostudománytól kezdve a beszéd- és arcfelismerésig számos területen alkalmazható.

A mesterséges intelligencia jelentős részét képezik a megoldáskereső rendszerek, melyek teljesítménye döntő szempont lehet bonyolultabb problémák megoldása során. Dolgozatom célja, hogy bemutasson néhány hatékony keresőalgoritmust, valamint ezek alkalmazását egy adott probléma esetén.

A dolgozat az alábbi módon épül fel. Az alapfogalmak bevezetése után egy híres optimalizálási feladatot, az utazó ügynök problémáját tárgyalom, majd ehhez több megoldáskereső módszert mutatok be. Ezután a lokális keresőalgoritmusokat, ezen belül a szimulált hűtést és a tabu-keresést fejtem ki.

A dolgozatban bemutatott algoritmusok működésének demonstrálása céljából elkészítettem a TSP Solver nevű Java-alkalmazást, mely az utazó ügynök problémáját képes megoldani néhány különböző módszer segítségével. A TSP elnevezés a Travelling Salesman Problem (az utazó ügynök problémája) rövidítéséből ered.

I. rész
Alapfogalmak

1. Állapottér-reprezentáció

Ahhoz, hogy meg tudjunk oldani egy problémát a mesterséges intelligenciában, szükségünk van arra, hogy a problémát valamilyen módon leírjuk. Erre több különböző technika létezik, azonban a legelterjedtebb módszer az állapottér-reprezentáció. Ennek során összegyűjtjük az adott probléma azon meghatározóit, melyek a problémát különböző értékekkel jellemzik, ezen jellemzők által felvett értékek összessége írja le a probléma egy állapotát. Állapottérnek nevezzük a probléma lehetséges állapotainak halmazát.

Tegyük fel, hogy a probléma elemzése során m meghatározó jellemzőt találtunk. Hogyha H_i jelöli az i. jellemző által felvehető értékek halmazát, akkor a probléma állapotai a

$$H_1 \times H_2 \times \ldots \times H_m$$

halmaz elemei lesznek. Azonban a fenti halmaz bővebb a probléma állapotainak halmazánál, tehát szükségünk van arra, hogy meghatározzuk, a fenti halmaz elemei közül melyek lesznek valóban állapotok. Ezt a kényszerfeltétel segítségével tehetjük meg. Az állapottér tehát a következő:

$$A = \{a | a \in H_1 \times H_2 \times \ldots \times H_m \text{ és kényszerfeltétel}(a)\}.$$

Szükség van arra, hogy meghatározzuk azt a kezdőállapotot, melyből a megoldáskeresés elindulhat. Kezdőállapotnak az állapottér azon elemét nevezzük, mely a probléma jellemzőinek kezdőértékét reprezentálja. Meg kell adnunk azokat az állapotokat is, melyek a probléma megoldását reprezentálják, ezeket célállapotnak nevezzük. A célállapotok halmazát megadhatjuk elemeinek explicit felsorolásával, vagy célfeltétel segítségével.

Ahhoz, hogy eljuthassunk egy célállapotba, bizonyos állapotokat meg kell tudnunk változtatni. Ezeket az állapotváltozásokat leíró leképezéseket operátoroknak nevezzük. Mivel nem biztos, hogy egy adott operátor minden állapotra alkalmazható, meg kell adnunk a leképezések értelmezési tartományát, melyet operátoralkalmazási előfeltételnek nevezünk.

Egy probléma állapottér-reprezentációja tehát akkor készült el, hogyha meghatároztuk a probléma állapotterét, kezdőállapotát, célállapotainak halmazát, valamint az operátorok halmazát.

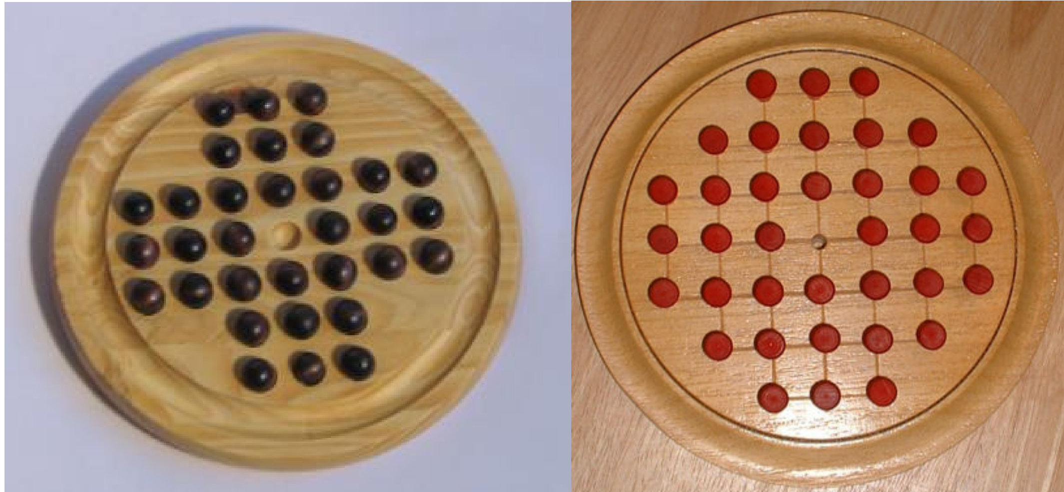
1.1. Példa: Peg Solitaire

1.1.1. A probléma leírása

A peg solitaire egy a XIX. századból származó egyszemélyes táblajáték. Egyes források szerint a játékot egy a Bastille-ba bebörtönzött francia arisztokrata alkotta meg, más források szerint a játékot amerikai indiánok találták ki, azonban egyik feltételezés sem bizonyított.

A játék lényege, hogy a táblán található bábuk (peg) szabályos lépésekkel egy kivételével lekerüljenek a tábláról. A megmaradt bábunak a tábla középén kell szerepelnie.

Szabályos lépésnek az számít, ha egy bábuval egy (vízszintesen vagy függőlegesen) szomszédos bábut átugrunk és azt a bábut, amit átugrottunk, levesszük a tábláról. Ugrás akkor lehetséges, hogyha az átugratni kívánt bábu túloldalán van szabad hely.



(a) „angol" játéktábla (b) „európai" játéktábla

1. ábra. Hagyományos játéktáblák

A játéktáblának két hagyományos változata létezik: az „angol" és az „európai" (1. ábra).

1.1.2. A probléma állapottér-reprezentációja

Állapottér. A probléma világának állapotait egy mátrix reprezentálja. Legyen

$$A = (a_{ij})_{7\times 7} = \begin{pmatrix} a_{11} & \dots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \dots & a_{77} \end{pmatrix},$$

ahol

$$a_{ij} = \{-1, 0, 1\}, \qquad i, j = 1, \dots, 7.$$

Az a_{ij} elem a játéktábla i. sorában és j. oszlopában található mezőt jellemzi:

- ha értéke -1, akkor az adott pozíció érvénytelen (oda nem léphetünk),
- ha értéke 0, akkor az adott pozíció érvényes és szabad (nincs rajta peg),
- ha értéke 1, akkor az adott pozíció érvényes és foglalt (van rajta peg).

Kezdőállapot. Két hagyományos kezdőállapotot különböztetünk meg:

- „angol” tábla esetén

$$k = \begin{pmatrix} -1 & -1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & -1 & -1 \end{pmatrix},$$

- „európai” tábla esetén

$$k = \begin{pmatrix} -1 & -1 & 1 & 1 & 1 & -1 & -1 \\ -1 & 1 & 1 & 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 & 1 & -1 & -1 \end{pmatrix}.$$

Célállapotok halmaza. A célállapotok halmaza a két esetben:

- „angol” tábla esetén

$$C = \begin{pmatrix} -1 & -1 & 0 & 0 & 0 & -1 & -1 \\ -1 & -1 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & -1 & -1 \\ -1 & -1 & 0 & 0 & 0 & -1 & -1 \end{pmatrix},$$

- „európai” tábla esetén

$$C = \begin{pmatrix} -1 & -1 & 0 & 0 & 0 & -1 & -1 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 0 & -1 & -1 \end{pmatrix}.$$

Operátorok. Négy különböző operátort definiálunk, ezek mindegyike a paraméterként megadott helyzetű bábuval egy adott irányba – balra, jobbra, fel, le – ugorja át a szomszédos bábut, valamint távolítja el az átugrott bábut.

A balra operátor. Ez az operátor az aktuális bábutól balra található bábut ugorja át és távolítja el a tábláról.

$$\text{balra}\colon \text{dom}(\text{balra}) \to A,$$

ahol

$$\text{dom}(\text{balra}) \subseteq A \times D \qquad D = \{1, \ldots, 7\} \times \{1, \ldots, 7\}.$$

Az operátor hatása:

$$\text{balra}(\begin{pmatrix} a_{11} & \dots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \dots & a_{77} \end{pmatrix}, (d_s, d_o)) = \begin{pmatrix} a'_{11} & \dots & a'_{17} \\ \vdots & \ddots & \vdots \\ a'_{71} & \dots & a'_{77} \end{pmatrix},$$

ahol

$$a'_{ij} = \begin{cases} 1, & \text{ha j = d}_o - 2 \land i = d_s \\ 0, & \text{ha (j = d}_o - 1 \land i = d_s) \lor (j = d_o \land i = d_s) \\ a_{ij}, & \text{egyébként} \end{cases}$$

i, j = {1, ..., 7}.

Az operátor alkalmazási előfeltétele:

$$d_o \geq 3 \land a_{d_s d_o - 2} = 0 \land a_{d_s d_o - 1} = 1 \land a_{d_s d_o} = 1.$$

A jobbra operátor. Ez az operátor az aktuális bábutól jobbra található bábut ugorja át és távolítja el a tábláról.

$$\text{jobbra}\colon \text{dom}(\text{jobbra}) \to A,$$

ahol

$$\text{dom}(\text{jobbra}) \subseteq A \times D \qquad D = \{1, \ldots, 7\} \times \{1, \ldots, 7\}.$$

Az operátor hatása:

$$\text{jobbra}(\begin{pmatrix} a_{11} & \dots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \dots & a_{77} \end{pmatrix}, (d_s, d_o)) = \begin{pmatrix} a'_{11} & \dots & a'_{17} \\ \vdots & \ddots & \vdots \\ a'_{71} & \dots & a'_{77} \end{pmatrix},$$

ahol

$$a'_{ij} = \begin{cases} 1, & \text{ha j = d}_o + 2 \land i = d_s \\ 0, & \text{ha (j = d}_o + 1 \land i = d_s) \lor (j = d_o \land i = d_s) \\ a_{ij}, & \text{egyébként} \end{cases}$$

i, j = {1, ..., 7}.

Az operátor alkalmazási előfeltétele:

$$d_o \leq 5 \land a_{d_s d_o + 2} = 0 \land a_{d_s d_o + 1} = 1 \land a_{d_s d_o} = 1.$$

A fel operátor. Ez az operátor az aktuális bábu fölött található bábut ugorja át és távolítja el a tábláról.

$$\text{fel}\colon \text{dom(fel)} \to A,$$

ahol

$$\text{dom(fel)} \subseteq A \times D \qquad D = \{1, \ldots, 7\} \times \{1, \ldots, 7\}.$$

Az operátor hatása:

$$\text{fel}(\begin{pmatrix} a_{11} & \ldots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \ldots & a_{77} \end{pmatrix}, (d_s, d_o)) = \begin{pmatrix} a'_{11} & \ldots & a'_{17} \\ \vdots & \ddots & \vdots \\ a'_{71} & \ldots & a'_{77} \end{pmatrix},$$

ahol

$$a'_{ij} = \begin{cases} 1, & \text{ha j = d}_o \wedge i = d_s - 2 \\ 0, & \text{ha (j = d}_o \wedge i = d_s - 1) \vee (j = d_o \wedge i = d_s) \\ a_{ij}, & \text{egyébként} \end{cases}$$

i, j = {1, ..., 7}.

Az operátor alkalmazási előfeltétele:

$$d_s \geq 3 \wedge a_{d_s - 2 d_o} = 0 \wedge a_{d_s - 1 d_o} = 1 \wedge a_{d_s d_o} = 1$$

A le operátor. Ez az operátor az aktuális bábu alatt található bábut ugorja át és távolítja el a tábláról.

$$le\colon \text{dom}(le) \to A,$$

ahol

$$\text{dom}(le) \subseteq A \times D \qquad D = \{1, \ldots, 7\} \times \{1, \ldots, 7\}.$$

Az operátor hatása:

$$le(\begin{pmatrix} a_{11} & \ldots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \ldots & a_{77} \end{pmatrix}, (d_s, d_o)) = \begin{pmatrix} a'_{11} & \ldots & a'_{17} \\ \vdots & \ddots & \vdots \\ a'_{71} & \ldots & a'_{77} \end{pmatrix},$$

ahol

$$a'_{ij} = \begin{cases} 1, & \text{ha j = d}_o \wedge i = d_s + 2 \\ 0, & \text{(ha j = d}_o \wedge i = d_s + 1) \vee (j = d_o \wedge i = d_s) \\ a_{ij}, & \text{egyébként} \end{cases}$$

i, j = {1, ..., 7}.

Az operátor alkalmazási előfeltétele:

$$d_s \leq 5 \wedge a_{d_s + 2 d_o} = 0 \wedge a_{d_s + 1 d_o} = 1 \wedge a_{d_s d_o} = 1.$$

2. Gráfelméleti alapfogalmak

A következő részben ismeretetésre kerülő probléma modellezéséhez, valamint a megoldási módszerek megértéséhez szükség van néhány gráfelméleti fogalom bevezetésére.

Legyenek E és $N \neq \emptyset$ diszjunkt halmazok, és legyen $\varphi : E \rightarrow N \times N$ leképezés. Ekkor a $G = (E, \varphi, N)$ hármast irányított gráfnak nevezzük. E elemei a gráf élei, N elemei pedig a gráf csúcsai. Egy csúcspont fokszáma a csúcspontba futó élek számával egyezik meg. A $G' = (E', \varphi', N')$ gráfot a $G = (E, \varphi, N)$ gráf részgráfjának nevezzük, hogyha $E' \subset E$, $C' \subset C$ és minden $e \in E'$ esetén $\varphi'(e) = \varphi(e)$.

Jelölje $N * N$ az N-beli elemekből álló rendezetlen párok halmazát:

$$N * N = \{(n_1, n_2) | n_1, n_2 \in N \text{ és a sorrend nem számít}\}$$

Ekkor a $G = (E, \varphi, N)$ hármast irányítatlan gráfnak nevezzük. A G gráf súlyozott gráf, amennyiben a gráf minden éléhez egy számértéket (súlyt) rendelünk. Az élsúlyok legtöbbször valós számok, de bizonyos esetekben tehetünk további megszorításokat (pl. csak pozitív élsúlyok megengedettek).

Legyen $G = (E, \varphi, N)$ egy irányítatlan súlyozott gráf, ahol $N = \{1, 2, \ldots, n\}$ és $E = \{(i, j) | i \in N, j \in N\}$. Az (i, j) élhez tartozó költséget jelöljük $c(i, j)$-vel. Útnak nevezzük élek egy $\{(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k)\}$ halmazát, ahol $\forall p, q(p \neq q \supset i_p \neq i_q)$. Körnek nevezzük élek egy $\{(i_1, i_2), (i_2, i_3), \ldots, (i_k, i_1)\}$ halmazát, ahol $\forall p, q(p \neq q \supset i_p \neq i_q)$. A C kört a G gráfban Hamilton-körnek nevezük, hogyha C a G gráf összes csúcsán áthalad, tehát $k = n$. A következő részben ismertetett probléma esetén a Hamilton-kör helyett az útvonal kifejezést használjuk. A G gráf Euler-köre olyan kör, mely G összes élét pontosan egyszer tartalmazza. E bármely S részhalmazát tekintve S hossza a következő:

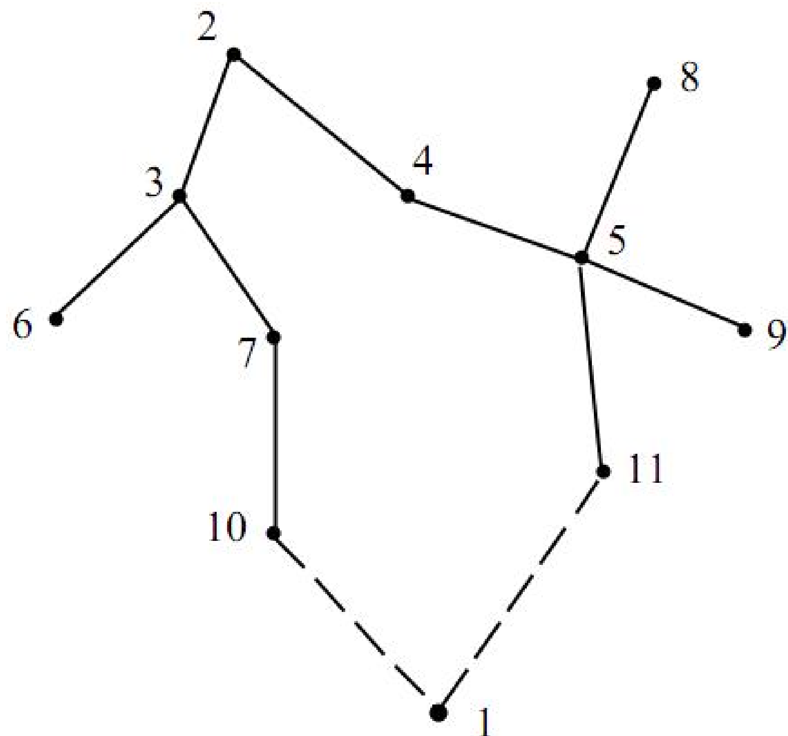
$$L(S) = \sum_{(i,j) \in S} c(i, j).$$

Optimális útvonalnak nevezzük a minimális hosszúságú útvonalat.

Egy gráf összefüggő, ha bármely két csúcsa között létezik út, azaz tetszőleges p és q csúcsra létezik $\{(p, i_1), (i_1, i_2), \ldots, (i_k, q)\}$. A fa olyan összefüggő gráf, mely nem tartalmaz kört.

Egy irányítatlan, súlyozott gráf feszítőfája a gráfnak az a részgráfja, amely fagráf és tartalmazza a gráf összes csúcspontját. A minimális feszítőfa minimális hosszúságú feszítőfa. A minimális feszítőfa nem feltétlenül egyértelmű, azonban súlya mindig az.

Bevezetjük az 1-fa fogalmát: a $G = (E, \varphi, N)$ gráf 1-fája olyan feszítőfa, mely tartalmazza az $N \backslash \{1\}$ halmazbeli csúcsokat, valamint az 1 csúcsba futó két élet. Az 1 csúcs tetszőlegesen megválasztható. Jegyezzük meg, hogy az 1-fa nem tekinthető fának, ugyanis tartalmaz kört (2. ábra). A minimális 1-fa minimális hosszúságú 1-fa.



2. ábra. 1-fa a speciális 1 csúccsal

II. rész
Az utazó ügynök problémája

Az utazó ügynök problémája a kombinatorikus optimalizálási problémák körébe tartozik. A probléma a következőképpen fogalmazható meg: adott n város és az út költsége bármely két város között, egy ügynöknek útja során az összes várost érintenie kell úgy, hogy minden városba csak egyszer jut el és az út végén visszatér a kiinduló városba. A feladat olyan útvonal meghatározása, melynek hossza (útköltsége) minimális, és teljesíti a fenti feltételeket.

A probléma pontos eredete máig sincs teljesen tisztázva. Egy utazó ügynököknek szóló 1832-es kézikönyv említést tesz a problémáról, sőt tartalmaz Németországra és Svájcra vonatkozó útvonalakat, azonban a probléma matematikai háttere ekkor még nincs megfogalmazva. Az utazó ügynök problémájával kapcsolatos matematikai feladatokkal először Sir William Rowan Hamilton és Thomas Penyngton Kirkman foglalkoztak az 1800-as években. A munkájukról szóló értekezés a Graph Theory című munkában található meg. Az utazó ügynök problémájának általános változatát először az 1930-as években vizsgálták, főleg Karl Menger. A problémával később Hassler Whitney és Merrill M. Flood is komolyan foglalkozott.

A probléma fontossága abban rejlik, hogy számos gyakorlati alkalmazás átültethető ebbe a formába (pl logisztikai folyamatok, mikrochipek gyártása, stb). Másrészt nagy jelentősége van a számításelmélet területén is, ugyanis a probléma bizonyítottan NP-nehéz, annak eldöntése pedig, hogy egy adott megoldás esetén létezik-e annál olcsóbb megoldása a konkrét esetnek, NP-teljes. Ez azt jelenti, hogy nem létezik olyan elég hatékony

algoritmus, mely megoldaná a problémát kellően nagy számú városra.

3. A probléma állapottér-reprezentációja

A probléma egy gráfelméleti feladatként értelmezhető: a gráf csúcspontjai a városokat, a gráf élei a városokat összekötő utakat, az élek súlyai pedig az egyes utak költségét reprezentálják. A feladat az, hogy megtaláljuk a gráfban lévő legrövidebb Hamilton-kört. Tehát a szimmetrikus utazóügynök-probléma a következőképpen fogalmazható meg: Határozzuk meg a G súlyozott gráfban található optimális útvonalat.

Alapvetően kétféle megközelítés létezik: a probléma szimmetrikus változata esetében egy adott él költsége mindkét irányban ugyanannyi, tehát ekkor irányítatlan gráfról van szó. Asszimmetrikus esetben egy adott él költsége eltérő a két irányból nézve, sőt az is elképzelhető, hogy csak az egyik irányban létezik út. Ekkor tehát irányított gráfról beszélhetünk. A továbbiakban az utazó ügynök problémája alatt a probléma szimmetrikus változatát értjük.

A probléma világának állapotai útvonalak, melyek meghatározzák, hogy milyen sorrendben járjuk be a városokat, azaz minden várost egy pozitív egész számmal jelölünk. Ezeket az állapotokat n elemű vektorok reprezentálják. Legyen

$$A = \{(a_1, a_2, \ldots, a_n) : a_i \in \{1, \ldots, n\}, i = 1, \ldots, n, \text{kényszerfeltétel}((a_1, a_2, \ldots, a_n)\}.$$

Az a_i elem reprezentálja az $i.$ várost az útvonalban.

A kényszerfeltétel:

$$\forall i, j(i \neq j \supset a_i \neq a_j),$$

tehát a vektorban az 1, ..., n értékek mindegyike pontosan egyszer fordul elő.

Első megközelítésben tekintsük kiinduló útvonalnak azt az útvonalat, melyben a városok az őket reprezentáló értékek szerint növekvő sorrendbe vannak rendezve, azaz

$$k = (1, 2, \ldots, n).$$

Később egyéb olyan módszerekről is lesz szó, melyek a kezdeti útvonal előállítására alkalmasak.

Célállapotnak azt az útvonalat tekintjük, melynek összköltsége (azaz az útban szereplő élek költségének összege) minimális, azaz ha $c_{i,j}$ jelöli az $i.$ városból a $j.$ városba vezető út költségét, akkor

$$C = \{(a_1, a_2, \ldots, a_n) \in A \text{ és célfeltétel}((a_1, a_2, \ldots, a_n))\},$$

ahol

$$\text{célfeltétel}((a_1, a_2, \ldots, a_n)) = \forall G(L(G) \geq L((a_1, a_2, \ldots, a_n))).$$

Itt G a feladat által meghatározott gráf egy Euler-köre, $L((a_1, a_2, \ldots, a_n))$ pedig az $a_1, a_2, \ldots, a_n$ csúcspontok által meghatározott gráf hossza.

Adott állapotból a k-opt lépéssel állíthatunk elő új állapotot, ezt használhatjuk operátorként. A k-opt lépés ismertetése a következő részben található.

4. Alapvető megoldási módszerek

4.1. A megoldás optimalitásának becslése

Célunk, hogy egy adott probléma esetén alsó korlátot adjunk a lehetséges útvonalak hosszára. Azaz egy olyan t értéket keresünk, melyre $l(T) \geq t$ bármely T útvonal esetén (az l függvény adja meg az útvonal hosszát). Először bevezetjük az 1-fa korlát fogalmát, majd ezt kiterjesztve eljutunk a pontosabb Held-Karp korlát fogalmához.

4.1.1. 1-fa korlát

Legyen (V, d) a problémát jellemző páros: V a csomópontok halmaza, d pedig a költségfüggvény. Egy adott problémapéldány 1-fa korlátja a következőképpen határozható meg:

1. Válasszunk egy $v_0 \in V$ csúcspontot.
2. Legyen r a $(V \backslash \{v_0\}, d)$ által meghatározott R^* feszítőfa hossza.
3. Legyen s a v_0-ból induló két legkisebb költséggel rendelkező él költségének összege, azaz

$$s = min\{d(v_0, x) + d(v_0, y) : x, y \in V \backslash \{v_0\}, x \neq y\}.$$

4. Legyen $t = r + s$.

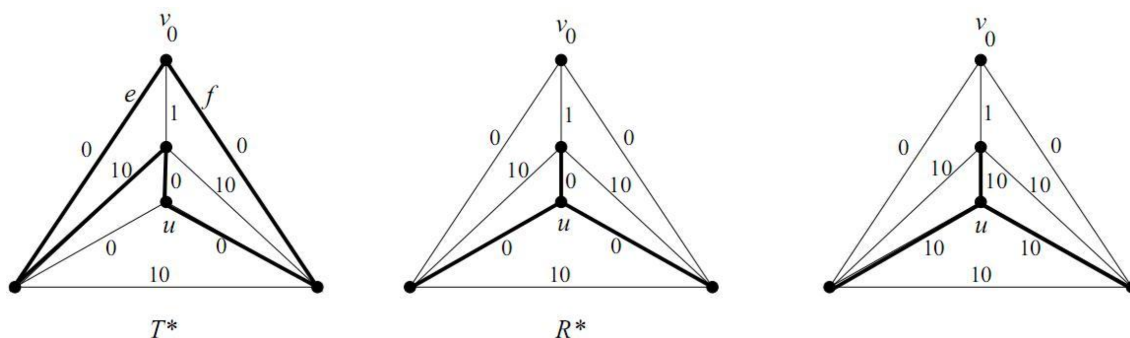
Minden T útvonalnak tartalmaznia kell a v_0-at tartalmazó két élet (e, f). Ha T-ből eltávolítjuk ezt a két élet, valamint v_0-t, akkor egy a $V \backslash \{v_0\}$ csúcspontokat tartalmazó R feszítőfát kapunk. Ekkor $l(T) = d(e) + d(f) + l(R) \geq s + l(R^*) = t$, tehát t a (V, d) párral leírt probléma alsó korlátja. Általában az 1-fa korlát 10%-kal az optimális útvonal l_{opt} értéke alatt található.

4.1.2. Held-Karp korlát

A Held-Karp korlát meghatározását egy példán keresztül mutatjuk be. A 3. ábrán látható módon az 1-fa korlát 0, pedig látható, hogy $l_{opt} = 10$. A problémát az okozza, hogy az R feszítőfában több 0 költségű él is szerepel. Tegyük fel, hogy az u csúcsból kiinduló élek mindegyikének költségét 10-zel megnöveljük. Minden útvonal a fenti élek közül pontosan kettőt tartalmaz, ezért az összes útvonal költsége 20-szal nő meg. Ekkor $l_{opt} = 30$. Másfelől az 1-fa korlát értéke 30-ra nőtt, és beláttuk, hogy T optimális. Ekkor azt mondjuk, hogy az u csúcsponthoz a -10 csúcspont-értéket rendeljük.

Általánosan hogyha az $u \in V$ csúcsponthoz hozzárendeljük a k csúcspont-értéket, akkor az u-ból kiinduló élek költségeit csökkentjük k-val. Tehát minden $u \backslash \{v_0\}$ csúcsponthoz hozzárendelhetjük az y_u csúcspont-értéket. Ezután minden $y = (y_u : u \backslash \{v_0\})$ vektorra meghatározzuk az R_y feszítőfát $(V \backslash \{v_0\}, d)$-re. Held-Karp alsó korlátnak nevezzük az

$$l(R_y) + 2 \sum_{u \in V \backslash \{v_0\}} y_u$$



3. ábra. A Held-Karp korlát meghatározása

értéket.

A Held-Karp korlátot különböző módszerek esetén gyakran százalékos formában adják meg. Értéke ekkor azt jelenti, hogy az adott eljárás által szolgáltatott megoldás milyen mértékben térhet el az optimális megoldástól.

4.2. Nyers erő (Brute force)

A legkézenfekvőbb megoldási módszer az lenne, ha végignéznénk az összes lehetséges útvonalat, majd ezek közül kiválasztanánk a legrövidebbet. Azonban mivel ez n város esetén n! permutációt jelentene, ez a módszer nagy n érték esetén kivitelezhetetlen. Tekintsünk erre egy példát! Legyen adott egy számítógép, mely másodpercenként 500000 műveletet képes elvégezni. 10 város esetén a gép 50000 utat tud végignézni egy másodperc alatt, és mivel 10!=3628800, így az optimális megoldás alig több, mint egy perc alatt megtalálható. Ez az idő 11 város esetén kb 13 perc, 13 város esetén közel 34 óra lenne.

Dinamikus programozási módszerek segítségével a megoldás lépésszáma $n^2 2^n$-nel felülbecsülhető. Ez még mindig n exponenciális függvénye, azonban hatékonyabb, mint az n! lépést végigvizsgáló brute force módszer.

4.3. Heurisztikus módszerek

A probléma megoldására számos olyan approximációs algoritmust és heurisztikát fejlesztettek ki, melyek extrém nagyságú feladatok (több millió város) esetén is képesek az optimális megoldástól nagy valószínűséggel csak 2-3%-kal eltérő megoldást szolgáltatni elfogadható időn belül.

4.3.1. Útvonal-konstrukció

Ezen algoritmusok célja egy a feltételeknek megfelelő útvonal előállítása. Az általuk szolgáltatott megoldáson a később ismertetésre kerülő útvonaljavító algoritmusok segítségével lehet javítani. A legjobb útvonal-konstrukciós algoritmusok általában 10-15%-kal térnek el az optimális megoldástól.

Random módszer. Ez a módszer egy teljesen véletlenszerű útvonalat készít, algoritmusa a következő:

```
1. Válasszunk véletlenszerűen egy kiinduló várost.
2. Jelöljük meg a választott várost (meglátogatva).
3. while (létezik nem megjelölt város) do
4.     Válasszunk véletlenszerűen egy várost az eddig meg
       nem látogatott városok közül.
5.     Jelöljük meg a választott várost (meglátogatva).
6.     Kössük össze a várost az előzőleg választott város-
       sal.
7. end while
```

Iteratív random módszer. Az előző megoldás „javított" változata: több útvonalat készít el a random módszer segítségével, majd ezek közül a legjobbat szolgáltatja.

```
1. while (nem teljesül a megállási feltétel) do
2.     Válasszunk véletlenszerűen egy kiinduló várost.
3.     Jelöljük meg a választott várost (meglátogatva).
4.     while(létezik meg nem jelölt város) do
5.         Válasszunk véletlenszerűen egy várost az eddig
           meg nem látogatott városok közül.
6.         Jelöljük meg a választott várost (meglátogatva).
7.         Kössük össze a várost az előzőleg választott vá-
           rossal.
8.     end while
9. end while
10. Az előállított útvonalakból válasszuk ki a legrövideb-
    bet.
```

Legközelebbi szomszéd (Nearest Neighbuor). Talán ez a legkézenfekvőbb heurisztika. Az algoritmus lényege, hogy adott városból mindig a legközelebbi várost látogatjuk meg. Bonyolultsága: $O(n^2)$, Held-Karp korlátja: 25%.

```
1. Válasszunk véletlenszerűen egy kiinduló várost.
2. Látogassuk meg az előző lépésben kiválasztott városhoz
   legközelebb fekvő, még meg nem látogatott várost.
3. if (létezik meg nem látogatott város) then
```

4.   Ugorjunk a 2. lépésre.

5. end if

6. Térjünk vissza a kiinduló városba.

Greedy-heurisztika. A greedy-heurisztika fokozatosan építi fel az útvonalat oly módon, hogy mindig kiválasztja a legrövidebb élet és hozzáadja az útvonalhoz, amennyiben ezáltal nem keletkezik n-nél kevesebb csúcsot tartalmazó kör, és egyik csúcs fokszáma sem lesz nagyobb 2-nél. Természetesen egy élet csak egyszer adhatunk hozzá az útvonalhoz. Bonyolultsága: $O(n^2 \log_2(n))$, Held-Karp korlátja: 15-20%.

1. Válasszuk ki az összes lehetséges él közül azt a legrövidebb élet, mely a fenti megszorításoknak eleget tesz.

2. if (nem n csúcsot tartalmaz az útvonal) then

3.   Ugorjunk az 1. lépésre.

4. end if

Beszúró heurisztikák (Insertion heuristics). A beszúró heurisztikák igen egyszerűek és több változatuk létezik. Ezen módszerek lényege, hogy kezdetben a városok halmazának egy részhalmazát tekintjük, majd a többi várost valamilyen heurisztika alapján illesztjük be. A kezdeti útvonal gyarkan egy háromszög, vagy konvex burok, esetleg kiindulhatunk magából egy élből is.

Legközelebbi beszúrása (Nearest Insertion), bonyolultsága: $O(n^2)$

1. Válasszuk ki a legrövidebb élet, tekintsük ezt a kezdeti részútvonalnak.

2. Válasszunk ki egy olyan várost, mely nem része a részútvonalnak és a részútvonal tetszőleges városához a legközelebb van.

3. A kiválasztott várost illesszük be a részútvonalba úgy, hogy a költségnövekedés minimális legyen.

4. if (van még olyan város, mely nem szerepel a részútvonalban) then

5.   Ugorjunk a 2. lépésre.

6. end if

Konvex burok (Convex Hull), bonyolultsága: O($n^2 \log_2(n)$)

1. `Határozzuk meg a kiválasztott részhalmaz konvex burkát, és képezzünk belőle egy részútvonalat.`
2. `A fenti részhalmazban nem szereplő városok mindegyikére határozzuk meg a legolcsóbb beillesztését (hasonló módon, mint az előző módszerben), majd válasszuk ki azt a várost, mely beillesztése a legkisebb költségnövekedéssel jár, és szúrjuk be.`
3. `if (van még olyan város, mely nem szerepel az útvonalban) then`
4. `Ugorjunk a 2. lépésre.`
5. `end if`

Christofides-heurisztika. A legtöbb heurisztika azt garantálja, hogy a legrosszabb eseben az optimális útvonal hosszának maximum kétszerese lesz a talált út hossza. Nicos Christofides professzor továbbfejlesztett egy ilyen algoritmust és azt tapasztalta, hogy a fenti arány itt csak $\frac{3}{2}$. Ez az algoritmus a Christofides-heurisztika néven ismert.

Az eredeti algoritmus a minimális feszítőfa fogalmán alapul, bonyolultsága: O($n^2 log_2(n)$).

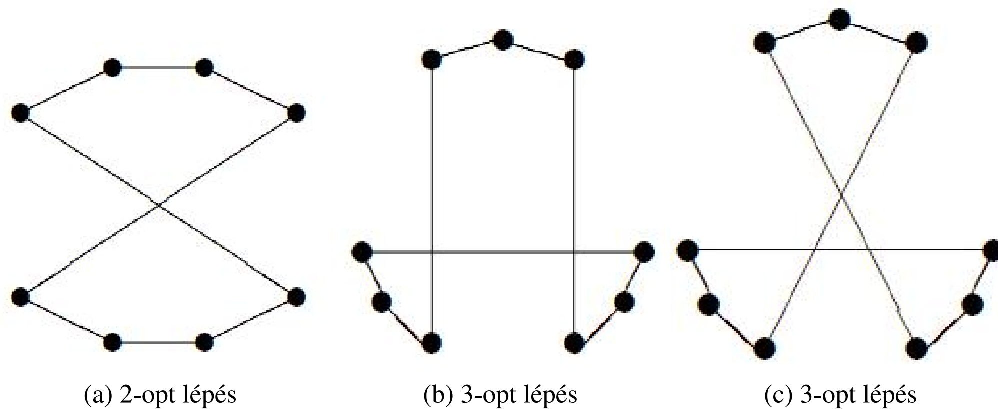
```
1. Határozzuk meg az összes várost tartalmazó halmaz mini-
   mális feszítőfáját.

2. Duplikáljuk az összes csúcspontot.

3. Készítsünk el egy Euler-kört.

4. Járjuk be a kört, de ne érintsük egyik csúcspontot sem
   egynél többször.
```

4.3.2. Útvonaljavítás

Ha az útvonal-konstrukciós algoritmusok által előállított megoldás nem felel meg elvárásainknak, javítanunk kell rajta. Ennek többféle módja létezik, azonban a legismertebbek a 2-opt és 3-opt lokális keresések. Ezek hatékonysága valamelyest függ attól, hogy milyen konstrukciós algoritmust használtunk az útvonal előállításához. Egy másik módszer a tabu-keresés, 2-opt és 3-opt lépések használatával. A szimulált hűtés nevű eljárás is ezeket a lépéseket használja működése során.

2-opt, 3-opt, k-opt. Egy k-opt lépés a következőt jelenti: az elkészült útvonalból kitörlünk k élet, majd az eredeti útvonal megmaradó részeit úgy probáljuk meg ismét összekötni, hogy a kezdetinél jobb megoldást kapjunk. Hogyha elvégezzük az összes lehetséges k-opt javítást, akkor azt mondjuk, hogy az úvonal k-optimális.

2-opt, 3-opt. A 2-opt algoritmus esetén 2-opt lépéseket alkalmazunk, ezek során tehát az útvonalból két élet törlünk, majd a két keletkezett részútvonalat ismét összekötjük. Ez a lépés csak egyféleképpen tehető meg úgy, hogy végül ismét a megszorításoknak eleget tevő útvonalat kapjunk (4. ábra). A „cserét” akkor végezzük el, hogyha ezáltal az új útvonal az eredetinél rövidebb lesz. Hogyha a 2-opt lépést addig alkalmazzuk, míg végül már ezzel nem érhető el több javítás, akkor eljutunk a 2-optimális útvonalhoz.



(a) 2-opt lépés (b) 3-opt lépés (c) 3-opt lépés

4. ábra. 2-opt és 3-opt lépések

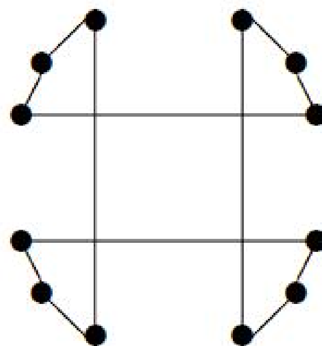
A 3-opt algoritmus a fentiekhez hasonlóképpen működik, azonban itt 2 él helyett 3 élet távolítunk el. Ez azt jelenti, hogy kétféleképpen tudjuk újra összekötni a keletkező három részútvonalat (4. ábra). Itt nem vesszük figyelembe azokat az eseteket, melyekhez egyetlen 2-opt lépéssel is el tudunk jutni. Az algoritmus akkor ér véget, hogyha nem végezhető el több 3-opt javítás. Hogyha egy útvonal 3-optimális, akkor egyben 2-optimális is.

Hogyha az útvonalra úgy tekintünk, mint városok egy permutációjára, akkor egy 2-opt lépés nem tesz mást, minthogy a permutáció egy szegmensének sorrendjét megfordítja. Hasonlóképpen egy 3-opt lépés két vagy három szegmens megfordítását jelenti.

k-opt. A 2-opt és 3-opt lépések mellett alkalmazhatunk 4-opt, 5-opt, stb. lépéseket, azonban ezek egyre több időt igényelnek, és csak kis mértékű javulást jelentenek a 2-opt és 3-opt heurisztikákhoz képest. Jegyezzük meg, hogy a 4-opt lépés nem bontható fel 2-opt lépések sorozatára (5. ábra).

5. A Lin-Kernighan algoritmus

A k-opt heurisztika hátránya, hogy k értékét előre meg kell adnunk, azonban ezt nehéz úgy megtennünk, hogy az algoritmus futási ideje, valamint a kapott megoldás megfelelő legyen. Lin és Kernighan azonban a változó k-opt algoritmus bevezetésével kiküszöbölte ezt a problémát. Az eljárás lényege az, hogy minden iteráció során újra meghatározza k értékét. Ez a következőképpen zajlik: ha az algoritmus éppen r él cseréjét vizsgálja, végrehajt egy tesztsorozatot, melyből kiderül, hogy $r + 1$ él cseréjét érdemes-e végrehajtani. Ezt a lépést addig folytatja, míg egy megadott végfeltétel be nem következik.



5. ábra. 4-opt lépés

Az algoritmus tárgyalása során először felvázoljuk az eredeti algoritmus működését, valamint a működéséhez szükséges feltételeket. Ezután megvizsgáljuk Lin és Kernighan finomításait, majd az algoritmus módosított változatát.

5.1. Az eredeti eljárás

Legyen T az aktuális útvonal. Az algoritmus minden iteráció során az éleknek egy olyan X=$\{x_1, \ldots, x_r\}$ és Y=$\{y_1, \ldots, y_r\}$ halmazát próbálja megtalálni, melyekre teljesül, hogy ha az X halmazbeli éleket töröljük a T útvonalból és ezeket helyettesítjük az Y halmazbeli élekkel, akkor ennek eredményeképpen egy jobb útvonalat kapunk. Az élek ilyen módon történő cseréit r-opt lépésnek hívjuk.

Az X és Y halmazok lépésről lépésre épülnek fel. Kezdetben X és Y üres. Az i. lépés során egy élpárt, x_i-t és y_i-t adjuk hozzá rendre az X és Y hlamazokhoz.

Annak érdekében, hogy az algoritmus eléggé hatékony legyen, érdemes bizonyos megszorításoknak tennünk az X és Y halmazokra.

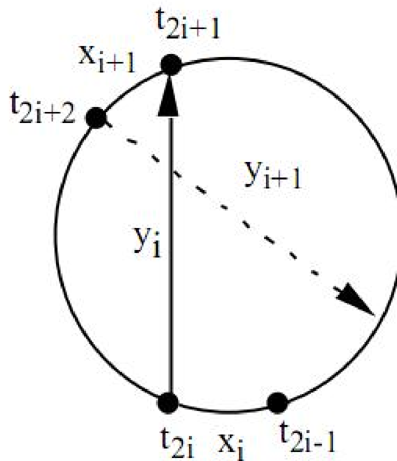
Szekvenciális csere-kritérium. Az x_i és y_i élek rendelkezzenek közös végponttal, ugyanez legyen igaz az y_i és x_{i+1} élekre is. Hogyha t_1 az x_1 valamely végpontja, akkor általánosan $x_i = (t_{2i-1}, t_{2i})$, $y_i = (t_{2i}, t_{2i+1})$ és $x_{i+1} = (t_{2i+1}, t_{2i+2})$, $i \geq 1$ (6. ábra).

Ezek alapján az $(x_1, y_1, x_2, y_2, \ldots, x_r, y_r)$ sorozat szomszédos élek láncát alkotja.

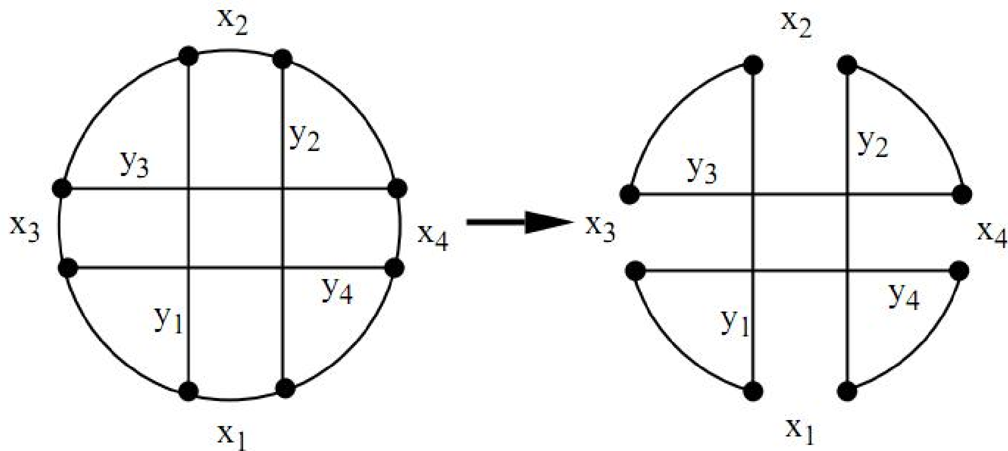
Annak szükséges (de nem elégséges) feltétele, hogy az X halmazbeli elemek Y halmazbeli elemekre történő cseréje útvonalat eredményezzen az, hogy a lánc zárt legyen, azaz $y_r = (t_{2r}, t_1)$. Az ilyen cseréket szekvenciálisnak nevezzük.

Általában egy útvonal javítása szekvenciális cserék alkalmazásával érhető el, a szóban forgó élek megfelelő számozása esetén. Azonban ez nem minden esetben van így. A 7. ábra olyan példát mutat, mely esetén szekvenciális csere nem alkalmazható.

Megvalósíthatósági kritérium. Szükség van arra, hogy $x_i = (t_{2i-1}, t_{2i})$ megválasztása úgy történjen, hogy ha t_{2i} t_1-hez csatlakozik, akkor az előálló konfiguráció útvonal legyen. A megvalósíthatósági kritériumot $i \geq 3$ esetén kell alkalmaznunk, és ez garantálja, hogy az útvonalat zárttá tudjuk tenni. Ez a megszorítás azért került bele az algoritmusba, hogy lerövidítse a futási időt, valamint egyszerűsítse a kódolást.



6. ábra. Szekvenciális csere-kritérium



7. ábra. Nem szekvenciális csere ($r = 4$)

Pozitív haszon kritérium. y_i-t minden esetben oly módon kell megválasztanunk, hogy G_i, a tervezett cserékből származó haszon pozitív legyen. Legyen $g_i = c(x_i) - c(y_i)$ az x_i y_i-re történő cseréjéből származó haszon. Ekkor $G_i = g_1 + g_2 + \dots, +g_i$. Ez a feltétel nagy szerepet játszik az algoritmus hatékonyságában. Az a követelmény, hogy minden G_i részösszeg pozitív legyen, talán túl szigorúnak tűnik. Azonban ez nem így van, ez a következő egyszerű állításból adódik: ha egy számsorozat tagjainak összege pozitív, akkor létezik ezen számoknak egy olyan ciklikus permutációja, melyben minden részösszeg pozitív.

Diszjunktivitási kritérium. Végül szükség van arra, hogy az X és Y halmazok diszjunktak legyenek. Ez leegyszerűsíti a kódolást, csökkenti a futási időt és hatékony megállási feltételként funkcionál.

5.1.1. Az algoritmus

Az algoritmus vázlatosan a következőképpen néz ki:

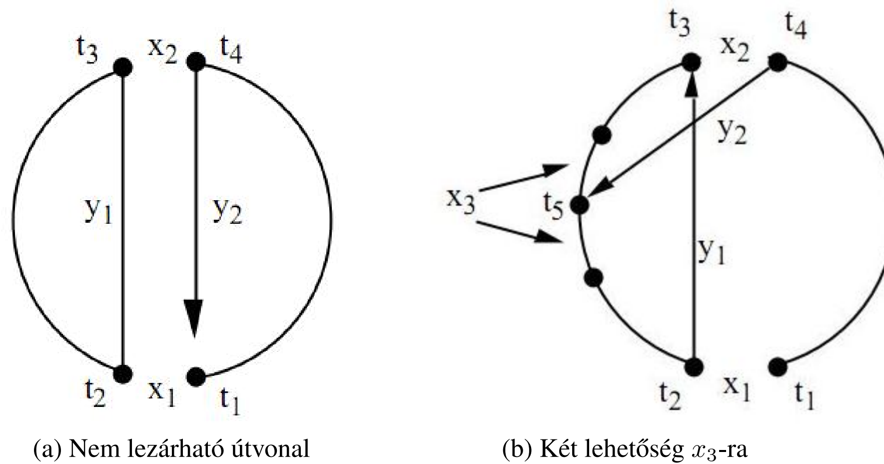
1. Legyen T egy véletlen kezdő útvonal.
2. Legyen $i = 1$, válasszuk ki t_1-et.
3. Válasszuk ki x_1-et, $x_1 = (t_1, t_2) \in T$.
4. if (létezik olyan y_1, melyre $G_1 > 0$ és $y_1 = (t_2, t_3) \notin T$) then
5. Válasszuk ki y_1-et.
6. else
7. Folytassuk a 39. lépésnél.
8. end if
9. if (létezik olyan y_1, melyre $G_1 > 0$, $y_1 = (t_2, t_3) \notin T$) then
10. Válasszuk ki y_1-et.
11. else
12. Ugorjunk a 39. lépésre.
13. Legyen $i = i + 1$.
14. Válasszuk ki x_i-t, $x_i = (t_{2i-1}, t_{2i}) \in T$, úgy, hogy teljesüljenek az alábbi követelmények:
    (a) ha t_{2i}-t csatlakoztatjuk t_1-hez, az így keletkező T' konfiguráció útvonal, és
    (b) $x_i \neq y_s$, hogyha $s < i$.
15. if (T' a T-nél jobb útvonal) then
16. Legyen T=T'.
17. Ugorjunk a 2. lépésre.
18. end if
19. Válasszuk ki y_i-t, $y_i = (t_{2i}, t_{2i+1}) \notin T$, úgy, hogy teljesüljenek az alábbi követelmények:
    (a) $G_i > 0$
    (b) $y_i \neq x_s$, ha $s \leq i$, és
    (c) x_{i+1} létezik.
20. if (létezik ilyen y_i) then
21. Ugorjunk a 13. lépésre.

```
22. end if
23. if (létezik y2-höz olyan alternatíva, melyet még nem pró-
    báltunk) then
24.     Legyen i = 2.
25.     Ugorjunk a 19.  lépésre.
26. end if
27. if (létezik x2-höz olyan alternatíva, melyet még nem pró-
    báltunk) then
28.     Legyen i = 2.
29.     Ugorjunk a 14.  lépésre.
30. end if
31. if (létezik y1-hez olyan alternatíva, melyet még nem pró-
    báltunk) then
32.     Legyen i = 1.
33.     Ugorjunk a 4.  lépésre.
34. end if
35. if (létezik x1-hez olyan alternatíva, melyet még nem pró-
    báltunk) then
36.     Legyen i = 1.
37.     Ugorjunk a 3.  lépésre.
38. end if
39. if (létezik t1-hez olyan alternatíva, melyet még nem pró-
    báltunk) then
40.     Ugorjunk a 2.  lépésre.
41. end if
42. Állj (vagy ugorjunk az 1.  lépésre).
```

Magyarázat az algoritmushoz. Az 1. lépés során egy véletlenszerű útvonalat választunk, az algoritmus ebből a kezdőállapotból indul ki.

A 3. lépés során kiválasztjuk az $x_1 = (t_1, t_2)$ élet. Miután kiválasztottuk t_1-et, x_1-re két lehetőségünk van. Itt kiválasztás alatt azt értjük, hogy egy eddig ki nem próbált alternatívát választunk. Azoban minden alkalommal, amikor az útvonalon javítást találunk (a 14. lépésben), az összes alternatívát ki nem próbáltnak tekintjük.

A 14. lépés során x_i kiválasztására két lehetőségünk van. Azonban adott y_{i-1} mellett ($i \geq 2$) ezek közül csak az egyik teszi lehetővé, hogy (y_i hozzáadásával) „lezárjuk" az útvonalat. Hogyha a másik lehetőséget választjuk, az két nem csatlakozó részútvonalat eredményez. Azonban csak egy esetben, nevezetesen $i = 2$ esetén megengedett ez a lehetőség (8/a ábra). Ha y_2-t úgy választjuk ki, hogy t_5 t_2 és t_3 közé esik, akkor az útvonal a következő lépésben lezárható. Azonban ekkor t_6 t_5 mindkét oldalán előfordulhat (8/b ábra): az eredeti algoritmus mindkét alternatívát megvizsgálja.



(a) Nem lezárható útvonal (b) Két lehetőség x_3-ra

8. ábra. Az algoritmus 6. lépése során lehetséges konfigurációk

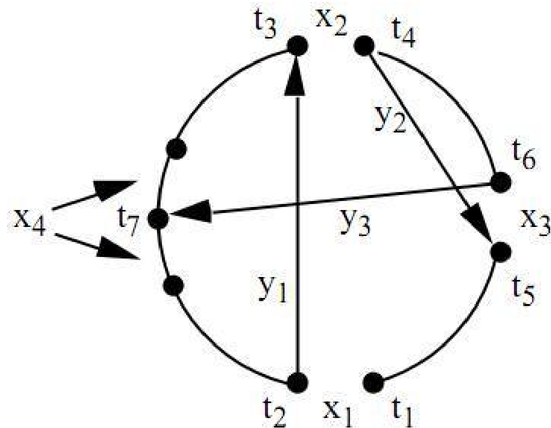
Másrészt hogyha y_2-t úgy választjuk meg, hogy t_5 t_4 és t_1 közé esik, akkor t_6 kiválasztására csak egy lehetőség adódik (t_4 és t_5 közé kell esnie), valamint t_7-nek t_2 és t_3 között kell elhelyezkednie. De ekkor t_8 t_7 mindkét oldalán elhelyezkedhet (9. ábra): az eredeti algoritmus azt az alternatívát vizsgálja meg, melyre $c(t_7, t_8)$ maximális.

A 14. és 19. lépésekben szereplő (b) feltétel azt biztosítja, hogy az X és Y halmazok diszjunktak: y_i nem lehet egy korábban elvetett él, x_i pedig nem lehet olyan él, melyet korábban már hozzáadtunk az útvonalhoz.

A 23. és 39. lépések visszalépést eredményeznek. Vegyük észre, hogy a visszalépés csak akkor engedélyezett, hogyha nem találtunk javítást.

A 42. lépésben az algoritmus megáll és eredményképpen egy útvonalat produkál, hogyha t_1 összes lehetséges értékét megvizsgálta és nem talált javítást. Hogyha szükséges, egy új véletlen útvonalból kiindulva újrakezdhetjük az algoritmust az 1. lépésnél.

A fenti algoritmus némileg eltér az eredeti algoritmustól. Itt ugyanis a T útvonalat azonnal helyettesítjük a T' útvonallal, amint javítást találtunk (14. lépés). Ezzel ellentétben az eredeti algoritmus folytatja a cseréket annak érdekében, hogy még rövidebb útvonalat találjon. Amennyiben már nincs több lehetőség cserére, vagy $G_i \leq G^*$, ahol g^* a T útvonal eddigi legjobb javítása, a keresés megáll, és T-t a legelőnyösebb útvonallal



9. ábra. x_3 egyértelmű, y_3-ra korlátozott számú lehetőségünk van, x_4 kétféle lehet

helyettesíti.

5.2. Lin és Kernighan finomításai

Az algoritmus szűk keresztmetszete az X és Y halmazokba kerülő élek meghatározása. A hatékonyság növelése érdekében tehát ezt a keresést lehetőleg minél inkább le kell rövidítenünk. Csak azokkal az élekkel kell foglalkozni, melyek nagy valószínűséggel az útvonal hosszának rövidülését eredményezik.

Az előző részben bemutatott algoritmus a következő feltételek segítségével limitálja a keresést:

A) Csak szekvenciális cserék megengedettek.

B) Az átmeneti haszonnak pozitívnak kell lennie.

C) Az útvonal „lezárható” (egy kivétellel, hogyha $i = 2$).

D) Egy korábban elvetett élet nem adhatunk az útvonalhoz, és egy korábban hozzáadott élet nem szakíthatunk meg.

Lin és Kernighan a következő finomítások bevezetésével igyekezett még inkább korlátozni a keresést [14]:

A) Az útvonalba bekerülő $y_i = (t_{2i}, t_{2i+1})$ él keresése t_{2i} öt legközelebbi szomszédjára szűkül.

B) $i \geq 4$ esetén egy x_i él nem vethető el, amennyiben az egy kisebb számú megoldás útvonalban szerepel.

C) A javítások keresése befejeződik, hogyha az aktuális útvonal megegyezik az előző útvonallal.

Az A) és B) szabályok heurisztikus szabályok, azon alapulnak, hogy mely élekkel kapcsolatban számítunk arra, hogy azok az optimális útvonalban részt fognak venni. E

két szabály segítségével csökkenthetjük a futási időt, de lehet, hogy emiatt nem találjuk meg az optimális megoldást.

A C) szabály szintén csökkenti a futási időt, azonban nem befolyásolja a megoldásokat. Hogyha egy útvonal megegyezik az előző lépésben megtalált útvonallal, nincs értelme megpróbálni tovább javítani. Ezzel megtakaríthatjuk azt az időtartamot, ami annak ellenőrzésével telne el, hogy végezhetünk-e további javításokat. Lin és Kernighan szerint ezzel átlagosan a futási idő 30-50%-át spóroljuk meg.

5.3. A módosított algoritmus

Az eredeti algoritmus kisebb számú város esetén igen hatékonynak bizonyult: annak valószínűsége, hogy megtalálja az optimális útvonalat, közel 100%. Azonban ez az arány több ezer város esetén jelentősen csökkent. Az algoritmus módosított, továbbfejlesztett változata nagy mértékű javulást mutat a hatékonyság terén.

5.3.1. Kiválasztott élek halmaza

Az eredeti algoritmus egyik központi eleme az a heurisztikus szabály, mely megszabja, hogy egy adott város esetén csak a hozzá legközelebb eső 5 városhoz vezető élek kerülhetnek be az útvonalba. Ez magában hordozza annak lehetőségét, hogy nem találjuk meg az optimális útvonalat, ugyanis előfordulhat, hogy az optimális útvonalban van olyan él, melynek végén szereplő városok egyike sincs a másik városhoz legközelebb eső 5 város között. Ez a probléma főleg nagyobb számú város esetén jelentkezik. A probléma kiküszöbölhető azzal, hogy a korlátnak 5-nél nagyobb értéket adunk, azonban ez jelentősen növeli az algoritmus futási idejét.

A fenti szabály azt sugallja, hogy minél rövidebb egy él, annál nagyobb annak valószínűsége, hogy az optimális útvonal tartalmazni fogja azt. Ezt kihasználva bevezetjük az α-mérték fogalmát, mely minimális feszítőfákat felhasználó érzékenységvizsgálaton alapul.

α-mérték. Látható, hogy

- az optimális útvonal egy olyan minimális 1-fa, melyben minden csúcs fokszáma 2,
- ha egy minimális 1-fa útvonal, akkor egyben optimális.

A probléma tehát a következőképpen fogalmazható meg: találjunk egy olyan minimális 1-fát, melyben minden csúcspont fokszáma 2.

Általában egy minimális feszítőfa sok olyan élet tartalmaz, melyek szerepelnek az optimális útvonalban. Az optimális útvonal általában a minimális 1-fában szereplő élek 70-80%-át tartalmazza. Tehát úgy tűnik, azok az élek, melyek részei a minimális 1-fának, nagy valószínűséggel az optimális útvonal részei is. Ennek megfelelően azon élek, melyek „távol állnak attól”, hogy a minimális 1-fa részei legyenek, kis valószínűséggel fordulnak elő az optimális útvonalban. A Lin-Kernighan algoritmusban ezen élek kizárhatók az útvonalat alkotó élek listájából. Ettől a lépéstől azt várjuk, hogy ne eredményezze azt, hogy az optimális útvonalat nem találjuk meg.

A fent említett „közelséget” a következőképpen írhatjuk le formálisan: legyen T egy $L(T)$ hosszúságú minimális 1-fa, és jelölje $T^+(i,j)$ azt a mininmális 1-fát, mely tartalmazza az (i,j) élet. Ekkor az (i,j) él α-mértéke $\alpha(i,j) = L(T^+(i,j)) - L(T)$. Azaz hogyha adott egy 1-fa hossza, akkor egy él α-mértéke azt jelenti, hogy mennyivel fog megnőni az 1-fa hossza, ha az élet beillesztjük az 1-fába.

Az α-mérték jellemzői:

- $\alpha(i,j) \geq 0$, és
- ha az (i,j) él része egy 1-fának, akkor $\alpha(i,j) = 0$.

Az α-mérték tehát arra használható, hogy azonosítsuk azokat az éleket, melyekről elképzelhetőnek tartjuk, hogy az optimális útvonal részei lesznek, a többi élet pedig figyelmen kívül hagyjuk. Ezen „ígéretes” élek által alkotott halmaz, a kiválasztott élek halmaza például állhat a csúcspontokhoz az α-mérték szerint legközelebb álló élekből, vagy pedig azon élekből, melyek α-mértéke egy meghatározott korlát alá esik.

Általában a kiválasztott élek halmazának meghatározásakor az α-mérték használata sokkal jobb választás, mint a legközelebbi szomszédok módszere. A halmaz általában kevesebb elemet tartalmaz, viszont ez nem hat negatívan a végeredményre.

Az α-mérték kiszámítása. Legyen $G = (N, E)$ egy teljes gráf. Keressük meg a G-hez tartozó mininális 1-fát. Ez úgy tehető meg, hogy megalkotjuk a {2,3,...,n} csúcsokat tartalmazó gráf minimális feszítőfáját (pl. Prim algoritmusával [2]), majd az 1 csúcsból kiinduló két legrövidebb élet hozzáadjuk. Ezután minden (i,j) élre számítsuk ki $\alpha(i,j)$-t.

Legyen T egy minimális 1-fa. T-ből a következőképpen adható meg az a $T^+(i,j)$ minimális feszítőfa, mely tartalmazza az (i,j) élet:

1. `if` ((i,j) `része` T`-nek) then`
2. $T^+(i,j)$ `megegyezik` T`-vel.`
3. `else if (az` (i,j) `él valamelyik végpontján szereplő csúcs az 1 csúcs) then`
4. $T^+(i,j)$`-t úgy kapjuk meg` T`-ből, hogy az 1 csúcsból induló leghosszabb élet kicseréljük` (i,j)`-vel.`
5. `else`

   `Illesszük be az` (i,j) `élet` T`-be. Így egy kör keletkezik` T `feszítőfájában.` $T^+(i,j)$ `úgy állítható elő, hogy a körből eltávolítjuk az` (i,j) `éltől különböző élek közül a legrövidebbet.`
6. `end if`

5.3.2. Alapvető lépések

Alapvető kérdés, hogy az r-opt lépések mely részhalmazát használjuk arra, hogy az adott útvonalból jobb útvonalat próbáljunk meg előállítani. Azt eredeti algoritmus olyan r-opt lépéseket enged meg, melyek átalakíthatók oly módon, hogy egy 2-opt vagy 3-opt lépést 2-opt lépések sorozata kövessen. A lépéseknek eleget kell tenniük a szekvenciális csere -és megvalósíthatósági kritériumoknak is. Ettől az általános sémától kétféle módon térhetünk el a 4-opt lépések miatt:

- speciális esetben az első lépés lehet egy szekvenciális 4-opt lépés, melyet 2-opt lépések követnek,
- ha az útvonalat szekvenciális lépésekkel már nem tudjuk tovább javítani, akkor használhatók nem szekvenciális 4-opt lépések is.

A módosított algoritmus azonban több ponton is változtatásokat eszközöl. A leglényegesebb módosítás, hogy az alapvető lépés a szekvenciális 5-opt lépés (tehát 5-opt lépések sorozata). Egy lépés végrehajtása azonban megszakad, hogyha azt vesszük észre, hogy az útvonalat lezárva jobb megoldást kapunk. Így az algoritmus biztosítja a 2-, 3-, 4-, illetve 5-optimalitást.

5.3.3. Kezdeti útvonalak

Az eredeti algoritmus egy véletlenszerű útvonalból indul ki. Lin és Kernighan eredményei azt mutatták, hogy az útvonal-konstrukciós heurisztikák feleslegesek, túl sok időt vesznek igénybe.

A módosított algoritmus különböző implenetcáióinak használata során azt tapasztalták, hogy a megoldásként kapott útvonal nem függ erősen a kezdeti útvonaltól. Azonban jelentős javulás érhető el a futási idő tekintetében, hogyha a kezdeti útvonalak közel optimálisak.

III. rész

Lokális keresőalgoritmusok

A lokális keresőalgoritmusok jellemzője, hogy a megoldások halmazán egy szomszédsági függvény mellett olyan megengedett megoldást keresnek, melynek nincs nála jobb szomszédja. Az optimális megoldás megtalálása általában nem garantálható, emiatt ezeket az algoritmusokat közelítő algoritmusoknak (approximation algorithms) nevezzük.

6. Lokális keresési problémák

Legyen adott egy kombinatorikus optimalizálási probléma, melynek legyen x egy példánya. Az x által meghatározott megengedett megoldások halmazát jelöljük $S(x)$-szel. Legyen f_x az optimalizálandó célfüggvény. Olyan s megengedett megoldást keresünk, melyre teljesül, hogy minimalizálási probléma esetén $\forall r \in S(x) : f_x(r) \geq f_x(s)$, illetve maximalizálási probléma esetén $\forall r \in S(x) : f_x(r) \leq f_x(s)$. Azon s^* megengedett megoldásokat, melyek teljesítik a fentiek közül a megfelelő feltételt, (globálisan) optimális megoldásoknak nevezzük.

A megengedett megoldások halmazán definiáljuk az N szomszédsági függvényt, mely minden megengedett megoldáshoz hozzárendeli azokat a megengedett megoldásokat, melyeket bizonyos értelemben közelinek tekintünk. Azaz hogyha s egy tetszőleges megengedett megoldás, akkor $N(s, x) \subseteq S(x)$.

Az N szomszédsági függvényt általában műveletek segítségével definiáljuk. Legyen M egy művelethalmaz. M elemei a műveletek, melyek segítségével megengedett megoldásokat módosíthatunk, így szomszédos megengedett megoldásokat állíthatunk elő. Minden $m \in M$ művelet olyan parciális függvénynek tekinthető, melynek értelmezési tartománya és értékkészlete is $S(x)$ egy-egy részhalmaza. Hogyha $M(s)$ jelöli az s megengedett megoldásra alkalmazható műveletek halmazát, akkor

$$N(s, x) = \{r \in S(x) | \exists m \in M(s) : r = m(s)\}.$$

A lokális keresési probléma azt jelenti, hogy egy olyan $s \in S(x)$ megengedett megoldást kell találnunk, amely lokálisan optimális, tehát minimalizálási probléma esetén $\forall r \in N(s, x) : f_x(r) \geq f_x(s)$, maximalizálási probléma esetén pedig $\forall r \in N(s, x) : f_x(r) \leq f_x(s)$, ahol f_x az x problémapéldány célfüggvényét jelöli.

Látható, hogy egy adott optimalizálási feladathoz több különböző szomszédsági függvényt is definiálhatunk, így ugyanazon problémából más-más lokális keresési problémát származtathatunk. Egy adott szomszédsági függvény mellett a lokálisan optimális megoldásokra nem feltétlenül teljesül, hogy globálisan is optimálisak, így a kiinduló feladatnak sem megoldásai. Azonban hogyha minden lokálisan optimális megoldás egyben globálisan optimális, akkor egzakt szomszédsági függvényről beszélünk. Egzakt szomszédsági függvény lehet például a következő: $N(s, x) \equiv S(x) - \{s\}$, tehát minden s megengedett megoldásnak az összes többi megengedett megoldás a szomszédságába esik. Azonban így semmit sem egyszerűsítettünk a kiinduló problémán. Ennél bonyolultabb módszerekre van szükség, melyek a konkrét problémák elemzése során állíthatók elő. Általában a probléma bonyolultsága miatt nem törekszünk arra, hogy az optimális megoldást megtaláljuk, megelégszünk egy elég jó megoldással.

6.1. A szomszédsági gráf

A szomszédsági függvény egy irányított gráfot feszít ki a megengedett megoldások között. Ezt a gráfot szomsédsági gráfnak hívjuk. Legfontosabb jellemzői a következők:

- Csúcsai a megengedett megoldásokkal címkézhetők.
- Legyenek s és r megengedett megoldások. Pontosan akkor vezet s-ből r-be irányított él, hogyha $r \in N(s, x)$ teljesül.
- A gráf egy vagy több maximális komponensből áll, ahol két csúcs, s és r pontosan akkor tartozik ugyanahhoz a komponenshez, hogyha létezik irányított út s-ből r-be és r-ből s-be.
- A gráf adott csúcsából nem feltétlenül érhető el minden más csúcs irányított élsorozat mentén.

6.2. A lokális keresés alapalgoritmusa

Adott egy kombinatorikus optimalizálási probléma, valamint a megengedett megoldásokon definiált N szomszédsági függvény, így egy lokális keresési problémát kapunk. Célunk az, hogy találjunk egy lokálisan optimális megengedett megoldást. A következőképpen érdemes eljárni:

```
1. Válasszunk ki egy kezdeti megengedett megoldást (a szom-
   szédsági gráf egy csúcsa), ezt tekintsük aktuális megen-
   gedett megoldásnak.
2. Találjunk egy olyan megengedett megoldást az aktuális
   megengedett megoldás szomszédságában, mely jobb az aktu-
   ális megengedett megoldásnál.
3. if (a 2.  lépés sikeres) then
4.     A 2.  lépésben talált megengedett megoldás lesz az
       aktuális megengedett megoldás.
5.     Ismételjük meg a 2.  lépést.
6. else
7.     Megállunk, mert az aktuális megengedett megoldás lo-
       kálisan optimális (pivotálási szabály).
8. end if
```

Elvárjuk, hogy a kezdeti megengedett megoldás keresése, valamint az aktuális megengedett megoldás szomszédságában keresendő jobb megengedett megoldás találása egyszerűbb probléma legyen, mint a kiinduló probléma, ugyanis ellenkező esetben nem egyszerűsítünk a feladaton. A második lépés hatékonysága függ a szomszédsági függvény megválasztásától. Tegyük fel, hogy N_1 és N_2 szomszédsági függvények, és teljesül,

hogy $N_1(s,x) \subseteq N_2(s,x)$, tehát az N_2 szomszédság bővebb. Ekkor előfordulhat, hogy $N_2(s,x)$-ben létezik olyan r megoldás, mely $N_1(s,x)$-ben nincs benne, és r jobb megengedett megoldás, mint bármely $N_1(s,x)$-beli. Azonban mivel N_2 szomszédsága bővebb, több idő és tár szükséges a szomszédságban való kereséshez. Így egy algoritmustervezési dilemmához jutottunk, melyre nincs általános érvényű válasz.

Az egyik leggyakrabban használt lokális keresési módszer az ún. „hegymászó módszer”, mely valós értékű függvények minimumának közelítésére alkalmas. Az algoritmus a következőképpen működik:

1. Válasszunk egy i kezdeti megoldást az S halmazból.
2. Keressük meg a legjobb j megoldást $N(i)$-ben ($\forall k \in N(i) : f(j) \leq f(k)$).
3. if ($f(j) \geq f(i)$) then
4.   Vége.
5. else
6.   Legyen $i = j$.
7.   Ugorjunk a 2. lépésre.
8. end if

Látható, hogy ez az eljárás az f függvény lokális minimumának megtalálása esetén áll meg.

A lokális keresésnek két gyenge pontját emelhetjük ki:

- nem tudjuk előre, hogy hány lépés szükséges egy lokálisan optimális megoldás megtalálásához,
- általában nem tudjuk, hogy az algoritmus megállása után a talált lokálisan optimális megoldás mennyivel rosszabb, mint a globálisan optimális megoldás.

Ezek miatt nehezebb problémák esetén olyan algoritmusokat használnak, melyek nem állnak meg az első lokálisan optimális megengedett megoldás megtalálása után, valamint az algoritmus lépésszáma is kézben tartható.

7. Szimulált hűtés

Az első bemutatásra kerülő lokális kereső algoritmus a szimulált hűtés nevű eljárás.

7.1. A módszer eredete, fizikai analógia

A szimulált hűtés módszerének eredete a következő fizikai eljáráshoz köthető: adott egy test, melyet olyan állapotba szeretnénk hozni, melyben a test részecskéi jól struktúrált kristályrácsba rendeződnek és a test belső energiája minimális (ezt az állapotot alapállapotnak hívjuk). Először a testet felmelegítjük, míg meg nem olvad, így a részecskék szabadon tudnak mozogni. Ezután a testet lehűtjük, melynek során a részecskék alapállapotba tudnak rendeződni. Ha a testet nem melegítjük fel eléggé, vagy a hűtés mértéke nem megfelelő, akkor az alapállapot elérhetetlenné válik (például hogyha egy folyékony anyagot túl gyorsan hűtünk le, az anyag részlegesen optimális állapotban szilárdul meg, hogyha pedig lassan hűtjük, az anyag kristályai minimális energiájú állapotba kerülnek).

A módszer tehát párhuzamot von a statisztikai mechanika és a kombinatorikus optimalizálás között:

- az anyag hűtése (minimális energiaszintre juttatása) párhuzamba állítható az optimalizálási probléma megoldásával,
- a mechanika sok szabadsági fokkal rendelkező rendszerek viselkedésével foglalkozik termális egyensúly esetén, véges hőmérséklet mellett, míg matematikai szempontból egy adott célfüggvény minimumának megkeresése a feladat sok változó mellett.

A fizikai példa és a szimulált hűtés módszere között Metropolis algoritmusa valósítja meg az átmenetet.

7.2. Metropolis algoritmusa

Egy fizikai rendszer energetikai szempontból nincs elszigetelve környezetétől, köztük energiaátadás történhet. Ezt az energiaátadást a T hőmérséklettel írjuk le. Minél nagyobb T, a környezet annál inkább hajlamos energiát átadni a rendszernek, és minél alacsonyabb a hőmérséklet, a környezet annál inkább próbálja alacsony energiájú szinten tartani a rendszert. Annak valószínűsége, hogy a rendszer E energiaállapotban van T környezeti hőmérséklet mellett, a Boltzmann-tényezővel, $e^{-\frac{E}{T}}$-vel arányos, tehát $P \sim e^{-\frac{E}{T}}$.

Metropolis algoritmusa esetén a lehetséges konfigurációk terét termális szempontból közelítjük meg oly módon, hogy az egyes konfigurációk közötti átmeneteket próbáljuk felderíteni. Legyen A és B két olyan állapot, melyek a Boltzmann-tényezővel arányos valószínűséggel fordulnak elő. Ekkor

$$\frac{P(A)}{P(B)} = \frac{e^{-\frac{E_A}{T}}}{e^{-\frac{E_B}{T}}} = e^{-\frac{E_A - E_B}{T}}$$

Metropolis a következő algoritmust javasolta a fenti relatív valószínűség elérésére [12]:

1. `Induljunk az` E_A `energiájú` A `állapotból, majd eszközöljünk valamilyen változtatást, melynek hatására előáll egy új` B `állapot.`

2. `Számítsuk ki az` E_B `értéket (ez általában` E_A`-tól csak kis mértékben tér el).`

3. `if (`$E_B < E_A$`) then`

4. `Fogadjuk el az új állapotot, mivel kisebb energiával rendelkezik.`

5. `end if`

6. `if (`$E_B > E_A$`) then`

7. `az új (magasabb energiájú) állapotot fogadjuk el` $p = e^{-\frac{E_A - E_B}{T}}$ `valószínűséggel. Ez azt jelenti, hogy magas hőmérséklet esetén haladhatunk a „rossz" irányba is, azonban ahogy csökken a hőmérséklet, úgy egyre inkább törekszünk arra, hogy a „környező" állapotokból a legalacsonyabb energiájút fogadjuk el.`

8. `end if`

7.3. A szimulált hűtés algoritmusa, hűtési ütemtervek

Metropolis algoritmusának általánosítása a Kirkpatrick-algoritmus [10], mely a keresés hatékonyságának növelése érdekében hűtési ütemtervet tartalmaz.

A szimulált hűtés algoritmusát 1983-ban fejlesztették ki bonyolult kombinatorikus optimalizálási problémák megoldására. A módszer lényege a következő: feltesszük, hogy a problémapéldány megengedett megoldásai megfelelnek egy fizikai rendszernek (egy testet alkotó részecskék lehetséges állapotainak). Egy megengedett megoldás költsége megegyezik az állapothoz tartozó energiával. A fizikai példában szereplő hőmérséklet mintájára az algoritmusban szerepel egy hőmérséklet (vagy vezérlési) paraméter. A módszer előnye, hogy képes elkerülni azt, hogy lokális minimumoknál megrekedjen. Az algoritmus olyan véletlenszerű keresést valósít meg, mely (minimalizálási probléma esetén) nem csak a célfüggvény értékének csökkenését eredményező változásokat enged meg, hanem olyanokat is, melyek a célfüggvény értékét növelik. Ez utóbbit $p = e^{-\frac{\delta F}{T}}$ valószínűséggel fogadja el, ahol δF a célfüggvényben történt változás, T pedig a hőmérséklet (vagy vezérlési) paraméter.

Az algoritmus váza a következőképpen néz ki:

1. `Legyen` s `egy kezdő megengedett megoldás` $S(x)$`-ből.`

2. `Legyen` T_0 `a kezdő hőmérséklet.`

3. `Legyen` L_0 `a kezdő folyamathossz.`

4. `Legyen` $k = 0$`.`

```
 5. repeat
 6.     for n = 1 to L_k
 7.         Legyen r ∈ N(s,x).
 8.             if (f(r) ≤ f(s)) then
 9.                 Legyen s = r.
10.             else
11.                 if exp((f(s)−f(r))/T_k) > uniform[0,1) then
12.                     Legyen s = r.
13.                 end if
14.             end if
15.     end for
16. Legyen k = k+1.
17. Legyen T_k a következő hőmérséklet k és T_{k−1} függvényében.
18. Legyen L_k a következő folyamathossz k és L_{k−1} függvényében.
19. until (megállási feltétel)
20. return s
```

Az algoritmus két fő részre bontható:

- kezdeti megoldás előállítása,
- fokozatos javítás.

A célfüggvényértékek nem alkotnak szigorúan monoton csökkenő sorozatot a megengedett megoldások halmazán, mivel bizonyos valószínűséggel előfordulhat, hogy a keresés az aktuális megengedett megoldásnál rosszabbal folytatódik. Ez lehetővé teszi azt, hogy az algoritmus egy lokálisan optimális megengedett megoldásról egy rosszabbra lépjen tovább, majd ezáltal néhány lépés megtétele után az összes eddiginél jobb megoldást találjon.

Az algoritmus menete a következő: egy kezdeti megengedett megoldásból indulunk. A T_0 kezdeti hőmérsékletet olyan magasnak érdemes választani, mely mellett az algoritmus eleinte nagyobb valószínűséggel halad „rossz” irányba, így megfelelő szomszédsági függvény mellett az összes megengedett megoldás elérhető lesz. Az L_0 kezdeti folyamathossz meghatározza, hogy a T_0 hőmérsékleten hány átmenet történhet.

Az aktuális megengedett megoldást követő megengedett megoldást az átmenetfüggvény határozza meg. A 7. lépésben az $N(s,x)$ halmazból választ egy tetszőleges r szomszédot, majd a 8-14. lépésekben alkalmazza rá az elfogadási feltételt. Ez két részből áll: hogyha r nem rosszabb, mint az aktuális megengedett megoldás (s), akkor elfogadja, és

ez lesz az aktuális megengedett megoldás. Egyébként a következőképpen jár el: generál egy véletlen számot a $[0, 1)$ intervallumban egyenletes eloszlás szerint, majd ezt hasonlítja össze az $exp(\frac{f(s)-f(r)}{T_k})$ értékkel, és ettől függően fogadja el a megoldást. Hogyha elfogadja a rosszabb megoldást, akkor azzal, különben a korábbi aktuális megengedett megoldással folytatja a keresést. Adott T_k hőmérsékleten L_k átmenet lehetséges.

Ezek után újabb hőmérséklet kerül meghatározásra. A hőmérséklet paraméter szerepe, hogy a keresés előrehaladtával az algoritmus egyre kisebb valószínűséggel fogadjon el az aktuálisnál rosszabb megoldást.

Az eljárás használatához az alábbi elemek meghatározása szükséges:

- a megengedett megoldások reprezentációja,
- a megoldásokban történő véletlen változtatások mikéntje (megengedett megoldások előállítása),
- a problémában szereplő függvények értékének meghatározása szolgáló eszköz,
- hűtési ütemterv: kezdő hőmérséklet és a hőmérséklet csökkentésének szabályai.

7.3.1. Megengedett megoldások előállítása

A lokális keresésről tárgyaltak értelmében egy megengedett megoldásból a szomszédsági függvény segítségével állíthatunk elő újabb megengedett megoldást (a szomszédsági függvény által meghatározott halmaz elemei közül választhatunk). Jelölje x_i a folyamat során előállított i. megengedett megoldást.

Egész változós problémák esetén az

$$x_{i+1} = x_i + u$$

formula segítségével állíthatunk elő megengedett megoldásokat, ahol az u vektor gyakran a $(-1, 1)$ intervallumban található véletlenszámokat tartalmaz.

Hogyha a megoldandó probléma folytonos változókat tartalmaz, legegyszerűbben az

$$x_{i+1} = x_i + Cu$$

módszerrel állíthatunk elő „próbamegoldásokat”, ahol az u vektor az előzőekhez hasonlóan a $(-1, 1)$ intervallumbeli véletlenszám, C pedig egy olyan konstans diagonális mátrix, mely meghatározza az egyes változókban megengedett maximális változás mértékét.

Vanderbilt és Louie (1984) az

$$x_{i+1} = x_i + Cu$$

módszert javasolta [15], ahol az u vektor a $(-\sqrt{3}, \sqrt{3})$ intervallumból veszi fel értékeit, így átlaguk és egységnyi szórásuk is 0, a Q mátrix pedig a lépések mértékének eloszlását határozza meg. Ahhoz, hogy az adott S kovariancia-mátrix segítségével véletlen lépéseket generáljunk, az

$$S = QQ^T$$

egyenletet kell megoldanunk például Cholesky-felbontással. A keresés előrehaladtával S-et frissítenünk kell, hogy megefelelő információt tartalmazzon a lokális topológiáról.

7.3.2. Megoldás-kiértékelés

A szimulált hűtés algoritmusa tartalmaz egy olyan célfüggvényt, mely a megengedett megoldások kiértékelésére szolgál. Ez az algoritmus szempontjából „fekete doboznak" tekinthető, az algoritmus számítási hatékonyságának növelése érdekében azonban arra kell törekednünk, hogy a kiértékelés erőforrásigényét minél inkább csökkentsük (hiszen számos alkalmazás esetén ez a leginkább költséges művelet).

7.3.3. Hűtési ütemtervek

A hűtési ütemterv megválasztása kritikus az algoritmus szempontjából, ugyanis ez befolyásolja leginkább annak teljesítményét. A kezdeti hőmérsékletnek elég magasnak kell lennie ahhoz, hogy a rendszer teljesen „megolvadjon", és a keresés előrehaladtával el kell érnie a „fagypontot".

A szimulált hűtés standard implementációja a csökkenő hőmérséklet mellett előálló véges hosszúságú Markov-láncokra (azaz lehetséges megoldások véletlen szekvenciáira) épül. A következő paramétereket kell megadnunk:

- kezdő hőmérséklet (T_0),
- végső hőmérséklet (T_f) vagy pedig egy megállási feltétel,
- az átmenetek száma (Markov-láncok hossza),
- a hőméréklet csökkentésére vonatkozó szabály.

Kezdő hőmérséklet. T_0-t úgy kell megválasztani, hogy annak átlagos valószínűsége (χ_0), hogy a célfüggvényt növelő megoldás elfogadásra kerül, 0.8 körül legyen [10]. Ekkor T_0 értéke probléma-specifikus, azonban becsülhető a következőképpen: egy kezdeti keresés során kiszámítjuk a célfüggvényben bekövetkező átlagos növekedést ($\delta \bar{f}^+$). Ekkor

$$T_0 = -\frac{\delta \bar{f}^+}{\ln \chi_0}.$$

Hogyha a célfüggvény a fent említett kezdeti keresés során megfigyelt változásának szórása σ_0, akkor a következő képlet használható: $T_0 = \sigma_0$ [17].

Végső hőmérséklet. Az algoritmus egyszerűbb implementációjában a következő módokon határozható meg a végső hőmérséklet:

- rögzítjük azon értékek halmazát, melyek hőmérsékletértékként előfordulhatnak,
- rögzítjük a keresés során előállítható megengedett megoldások számát.

A keresés megállítható akkor is, hogyha már nem képes továbbhaladni. Ezalatt például a következőket érthetjük:

- egy adott hőmérséklet mellett előálló Markov-láncon belül nem figyelhető meg javulás (nem keletkezik újabb legjobb megoldás),
- a megoldás elfogadási aránya (a célfüggvényt növelő és csökkentő elfogadott megoldások aránya) egy adott kis χ_f érték alá esik.

Markov-láncok hossza. A k-adik Markov-lánc hosszát (L_k) úgy érdemes megválasztani, hogy az a probléma méretétől függjön, így L_k független legyen k-tól. Emellett úgy is gondolkodhatunk, hogy minden hőmérséklet mellett legyen egy minimális számú megoldás (η_{min}), melyet elfogadunk. Azonban ahogy T_k közelít a 0-hoz, a megoldásokat egyre csökkenő valószínűséggel fogadjuk el, így az η_{min} számú elfogadás teljesüléséhez szükséges megoldások száma a végtelenhez tart. Tehát gyakorlati szempontból az algoritmus működése során egy Markov-lánc L_k próba vagy η_{min} elfogadás után ér véget; amelyik a kettő közül előbb történt, azt fogadjuk el. Tipikusan $\eta_{min} \approx 0.6\mathrm{L}_k$.

A hőmérséklet csökkentése. A hőmérséklet csökkentésére vonatkozó legegyszerűbb szabály az exponenciális hűtési szabály (ECS):

$$T_{k+1} = \alpha T_k,$$

ahol α egy 1-nél kisebb, de 1-hez közeli értékű konstans. Ezt a módszert először Kirkpatrick javasolta $\alpha = 0.95$ mellett [10].

Többen olyan hűtési sémát javasoltak, melyek bizonyos értelemben adaptívak, azaz az algoritmus aktuális teljesítményéről származó statisztikai adatoktól függően módosítják az algoritmus paramétereit. Egy viszonylag egyszerű adaptív módszer a következő:

$$T_{k+1} = \alpha_k T_k,$$

$$\alpha_k = max[0.5, exp(-\frac{0.7T_k}{\sigma_k})],$$

ahol σ_k a T_k hőmérséklet mellett elfogadott megoldásokhoz tartozó célfüggvényértékek standard szórása [7].

Újraindítás. Amennyiben az algoritmus már nem halad előre, azaz az iteráció többszöri lefutása után sem talált a legutóbb felfedezett legjobb megoldásnál jobb megoldást, akkor hatékonynak bizonyulhat a következő stratégia: a hőmérséklet változatlanul tartása mellett indítsuk újra a keresést az eddig talált legjobb megoldástól. Ezt a stratégiát azonban óvatosan kell használnunk, ugyanis hogyha a keresés újraindításához szükséges feltételek túl könnyen teljesülnek, akkor a keresési térnek csak egy kis része (valószínűleg egy lokális minimum) lesz felfedezve.

7.4. Előnyök, hátrányok

A szimulált hűtés képes magas fokú nemlineáris modelleket, kaotikus és zajos adatokat kezelni, meglehetősen robusztus módszer. Fő előnye a többi lokális keresőmódszerrel szemben a rugalmassága, valamint az, hogy képes a globálisan optimális megoldáshoz eljutni. Rugalmassága abban rejlik, hogy maga a módszer nem függ a modelltől. Mindezek mellett a módszer könnyen hangolható (pl. megfelelő hűtési ütemterv megválasztásával).

Azonban mivel a szimulált hűtés metaheurisztika, sok egyéb teendőt kell elvégeznünk, hogy valódi algoritmust készítsünk belőle. A paraméterek megfelelő hangolása szintén időigényes lehet. Implementáció szintjén pedig a felhasznált számok pontosságának mértéke nagy hatással lehet az eredményre.

7.5. Alkalmazási területek, változatok

A szimulált hűtés egyik legnagyobb sikerű alkalmazási területe az integrált áramkörök (VLSI) tervezése, ezen belül az elhelyezési és huzalozási problémák. A módszer eredményesnek bizonyult az orvostudományon belül a tomográfiai eljárással készült felvételek feldolgozásában (Sundermann, 1995). Szimulált hűtés segítségével bonyolultabb gráfszínezési problémák is megoldhatók. Ez azért lényeges, mert számos olyan probléma létezik, mely gráfszínezésre vezethező vissza. Adott erőforrások megosztásának optimalizálása is ilyen probléma (például egy processzor regisztereinek változókhoz való hozzárendelése; ennek segítségével egy magasszintű programozási nyelv fordítóprogramja hatékonyabban képes optimalizálni a kódot). Ezeken kívül számos kombinatorikus optimalizálási problémára (négyzetes hozzárendelés, gráf-particionálás, stb.) sikeresen alkalmazták a módszert.

A szimulált hűtés algoritmusának egy változata az adaptív szimulált hűtés. Ez a módszer abban különbözik a szimulált hűtéstől, hogy az algoritmus paraméterei az aktuális állapotot figyelembe véve változnak az eljárás működése során. A módszer előnye, hogy adaptivitása miatt kevesbé érzékeny a felhasználó által elvégzett beállításokra.

7.5.1. A módszer alkalmazása az utazó ügynök problémája esetén

Ebben az esetben az útvonal hossza (L) reprezentálja az energiát, és a hőmérséklet paraméter dimenziója a hosszúság. Az implementáció során a következő két dologra kell figyelnünk:

- az adott A állapotból milyen módon állítjuk elő az új B állapotot,
- milyen hűtési ütemtervet alkalmazzunk.

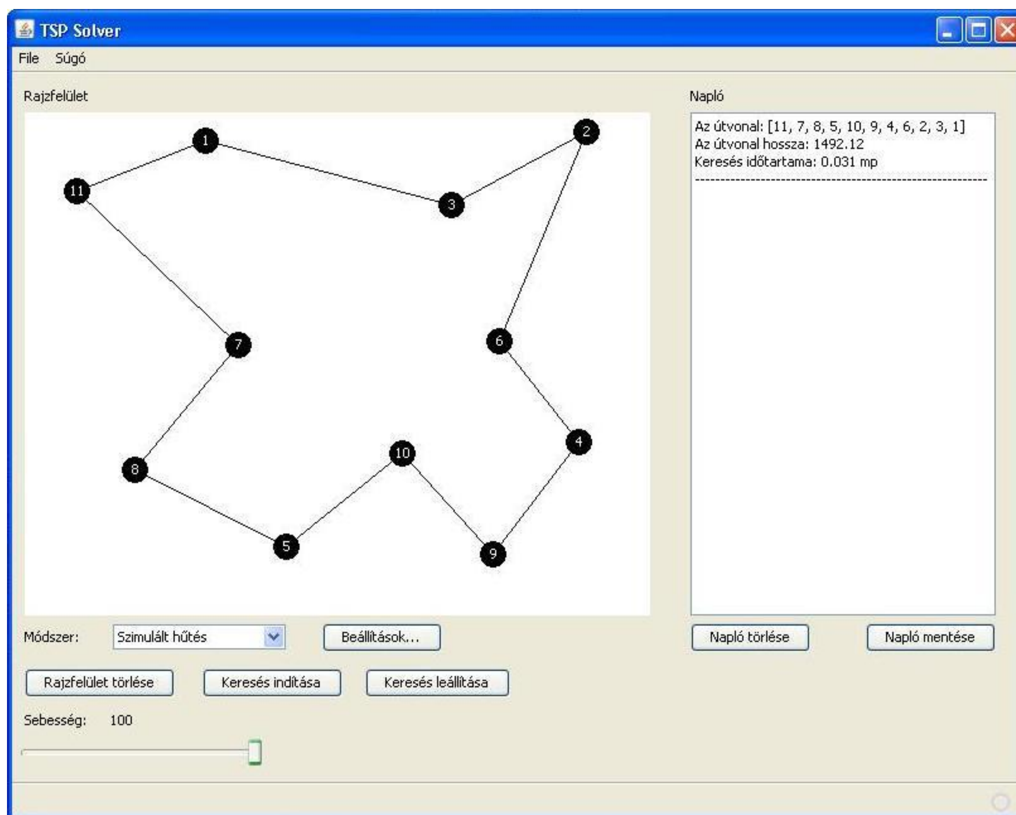
Az új állapot előállítása a következőképpen történhet:

- az útvonal egy részét eltávolítjuk és ugyanezt a részútvonalat fordított sorrendben beillesztjük az útvonalba, vagy
- az útvonal egy részét áthelyezzük az útvonal egy másik részére.

A 10. ábrán az algoritmus által talált útvonalat láthatjuk 11 város esetén.

8. Tabu-keresés

A következő algoritmus a lokális keresés algoritmusának egyszerű módosítása, kiegészítése. A szimulált hűtéshez hasonló modon ebben az esetben is jól kézbentartható az algoritmus lépésszáma, illetve az egyes lépések számítási igénye. Azonban a szimulált hűtéssel ellentétben a tabu-keresés esetében nincs általános elméleti magyarázat a gyakorlati sikerre.



10. ábra. A szimulált hűtés eredménye 11 város esetén

8.1. A módszer alapötletei

Annak érdekében, hogy javítani tudjunk a megoldások feltárásának hatékonyságán, nem csak lokális információt (a célfüggvény lokálisan optimális értékét) kell számon tartanunk, hanem szükség van a megoldások feltárásának folyamatával kapcsolatos információra is. A memória ily módon történő szisztematikus felhasználása a tabu-keresés egyik központi jellemzője. Míg a legtöbb megoldáskereső módszer csak f_{i^*} értékét tárolja a memóriában (ahol f a célfüggvény, i^* pedig a keresés során az adott pillanatig talált legjobb megoldás), addig a tabu-keresés a legutóbb talált megoldástól vezető utat is számon tartja. Ezt az információt arra használja fel, hogy kiválassza azt a $j \in N(i)$ megengedett megoldást, melybe i-ből lépni fog. A memóriának tehát az a szerepe, hogy az előbbi lépést $N(i)$ egy részhalmazára korlátozza úgy, hogy bizonyos szomszédokra történő lépést megtilt.

A legtöbb esetben nem garantált, hogy egy ilyen i^* megoldást megtalálunk, így a tabu-keresés egy általános heurisztikus eljárásnak tekinthető. Pontosabban fogalmazva a tabu-keresés egy olyan metaheurisztika, melynek feladata, hogy egy másik megoldáskereső eljárás működését irányítsa.

Első közelítésben a klasszikus „hegymászó” módszer átfogalmazásával nyerjük a tabu-keresés algoritmusát:

1. `Válasszuk ki az` i `kezdeti megoldást` S`-ből.`
2. `Állítsuk elő` $N(i)$ `egy` V^* `részhalmazát.`

3. `Keressük meg a legjobb` j `megoldást` V^*`-ban (azaz` $\forall k \in V^*$ `:` $f(j) \leq f(k)$`).`
4. `Legyen` $i = j$`.`
5. `if (`$f(j) \geq f(i)$`) then`
6. `    Vége.`
7. `else`
8. `    Ugorjunk a 2. lépésre.`
9. `end if`

Az eredeti „hegymászó” módszerben úgy járhatnánk el, hogy teljesül a $V^* = N(i)$ egyenlőség. Azonban ez meglehetősen időigényes lenne, tehát V^* megfelelő megválasztása nagy jelentőséggel bír. A másik véglet pedig $|V^*| = 1$, ez azonban j lehetséges értékeit erősen korlátozza.

Bizonyos speciális esetektől eltekintve a „hegymászó” módszerek használata általában nem kifizetődő, ugyanis valószínűleg egy lokális minimumnál meg fognak akadni, amely feltehetőleg elég távol áll a globális minimumtól. Egy iteratív feltáró eljárás esetén tehát szükség van arra, hogy bizonyos esetekben olyan lépést is tegyünk, mely a megoldáson nem javít. A szimulált hűtéssel ellentétben a tabu-keresés e legjobb j megoldást választja V^*-ból.

Azáltal, hogy megengedünk olyan lépéseket, melyek nem javítanak a megoldáson, fennáll annak a lehetősége, hogy egy már korábban meglátogatott megoldáshoz újból eljutunk, azaz jelen lesz a ciklikusság. A memóriának abban van szerepe, hogy megtiltja azon lépéseket, melyek nemrég meglátogatott megoldásokhoz vezetnének. A memória bevezetése miatt $N(i)$ helyett az $N(i, k)$ jelölést érdemes használnunk, ugyanis $N(i)$ elemei függenek az eddig meglátogatott megoldásoktól, így a k. iterációtól. Ezen jelölések birtokában a következőképpen fogalmazható meg az algoritmus (i^* az eddigi legjobb megoldás, k pedig a ciklusszámláló):

1. `Válasszunk egy kezdeti` i `megoldást` S`-ből.`
2. `Legyen` $i^* = i$`.`
3. `Legyen` $k = 0$`.`
4. `Legyen` $k = k + 1$`.`
5. `Állítsuk elő az` $N(i,k)$ `egy` V^* `részhalmazát.`
6. `Válasszuk ki a legjobb` V^*`-beli` j `megoldást.`
7. `Legyen` $i = j$`.`
8. `if (`$f(i) < f(i^*)$`) then`
9. `    Legyen` $i^* = i$`.`

```
10. end if
11. if (teljesül valamelyik megállási feltétel) then
12.     Vége.
13. else
14.     Ugorjunk a 4. lépésre.
15. end if
```

Megállási feltételek lehetnek a következők:

- $N(i, k+1) = \emptyset$,
- k nagyobb a megengedett iterációk számánál,
- a legutóbbi javítás óta végrehajtott iterációk száma nagyobb, mint egy adott érték,
- bizonyítottan megtaláltuk az optimális megoldást.

$N(i, k)$ definíciójából következik, hogy $N(i)$-ből eltávolítunk bizonyos nemrég megtalált megoldást, ezek tabu-megoldásoknak számítanak, melyeket figyelmen kívül kell hagynunk a következő iterációban. Így a memória használata részben megakadályozza, hogy a megoldáskeresés során kör alakuljon ki. Például hogyha a k. iteráció során a T lista (tabu-lista) tartalmazza a legutóbb meglátogatott $|T|$ megoldást, akkor a maximum $|T|$ hosszúságú köröket elkerülhetjük. Ebben az esetben tehát $N(i, k) = N(i) \backslash T$. Azonban nem biztos, hogy minden esetben érdemes a T listát tárolni, ezért másképpen közelítjük meg a dolgot.

Definiáljuk minden $i \in S$ megoldásra az $M(i)$ halmazt, mely azokat a lépéseket tartalmazza, melyek i-re alkalmazhatók annak érdekében, hogy előállítsuk az új j megldást. Ezt a következőképpen jelöljük: $j = i \oplus m$. Így $N(i) = \{j | \exists m \in M(i) : j = i \oplus m\}$. Olyan lépéseket használunk, melyek invertálhatók: minden m-hez létezik m^{-1}, melyre $(i \oplus m) \oplus m^{-1} = i$. Tehát ahelyett, hogy a T listában a legutóbb meglátogatott $|T|$ megoldást tárolnánk, a legutóbb megtett $|T|$ lépést, vagy pedig a legutóbb megtett lépések inverzét tartjuk számon. Látható, hogy ezzel a módosítással információt vesztünk: nincs arra garancia, hogy nem fog előfordulni maximum $|T|$ hosszúságú kör.

A hatékonyság érdekében érdemes lehet egyszerre több T_r listát számontartani. Ekkor bizonyos t_r komponensek tabu státuszt kapnak, ezzel jelezve, hogy jelenleg nem vehetnek részt egy lépésben. Általánosabban egy lépés tabu státusza egy olyan függvény, mely a komponenseinek tabu státuszára épül. Így

$$t_r(i, m) \in T_r, \qquad r = 1, \ldots, t.$$

Az i megoldásra alkalmazott m lépés tabu lépésnek számít, hogyha az összes feltétel teljesül.

Azáltal, hogy ily modón leegyszerűsítettük a tabu-listát (a megoldások helyett lépéseket tárolunk), előfordulhat, hogy olyan megoldások kapnak tabu státuszt, melyeket eddig még nem látogattunk meg. Így arra kényszerülünk, hogy enyhítsünk a tabu státuszon:

abban a esetben, hogyha a megoldás biztatónak tűnik, felülbírálhatjuk a tabu-státuszt. Ezt az ún. aspirációs szintekkel oldjuk meg.

Előfordulhat, hogy az i megoldásra érdemes alkalmazni az m tabu lépést, mert ezáltal az eddigi legjobb megoldásnál is jobb megoldást érünk el. Tehát az m lépést alkalmazni szeretnénk tabu státusza ellenére: ezt akkor tehetjük meg, hogyha az aspirációs szintje, $a(i,m)$ jobb, mint az $A(i,m)$ küszöbérték. $A(i,m)$ az $a(i,m)$ függvény által felvehető kedvező értékek halmazának tekinthető. Így az aspirációs feltételek a következő formában írhatók fel:

$$a_r(i,m) \in A_r(i,m), r = 1, \ldots, a.$$

Hogyha ezek közül a feltételek közül valamelyiket teljesíti az i megoldásra alkalmazandó m lépés, akkor m-et elfogadjuk státusza ellenére.

8.2. A memória effektív használata

Ahogy azt már korábban is említettük, a tabu-keresés egyik központi eleme a memória szisztematikus használata. A következőkben megvizsgáljuk a módszer azon elemeit, melyek a memória hatékony alkalmazására épülnek.

8.2.1. Változó hosszúságú tabu-lista

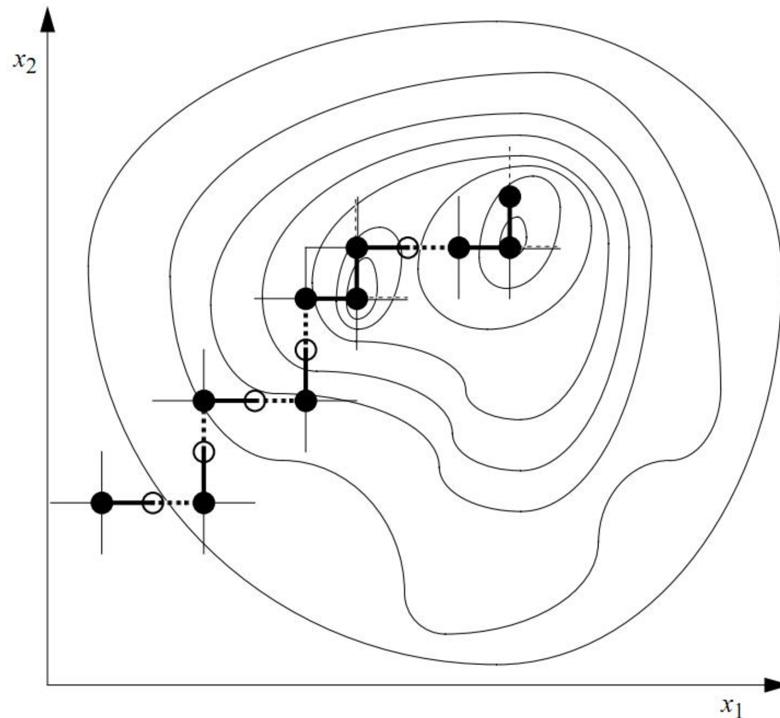
Láthattuk, hogy érdemes lehet párhuzamosan több listát is használni. A továbbiakban azt az esetet vesszük figyelembe, amikor egy listával rendelkezünk, azonban ez könnyen kiterjeszthető általánosabb esetekre is.

A tabu-lista alapvető feladata, hogy megakadályozza a körök kialakulását. Hogyha a lista hossza túl kicsi, akkor nem biztos, hogy el tudja látni feladatát; hasonlóképpen túl nagy méretű lista esetén túl sok megszorítás keletkezik. Megfigyelték, hogy a tabu-lista méretének növelése következtében a meglátogatott megoldások átlagos száma nő. Azoban egy adott optimalizálási probléma esetén gyakran nagyon nehéz, vagy egyenesen lehetetlen olyan értéket találni, amely mellett nem keletkeznek körök.

Ezeket a nehézségek leghatékonyabban változó hosszúságú tabu-listával oldhatjuk meg. Ekkor minden elem egy maximális és minimális érték által korlátozott számú iteráción keresztül tartozik a listához.

8.2.2. Rövidtávú memória

Az algoritmus a rövidtávú memóriát használja arra, hogy a keresés múltjáról információt tároljon. Itt található a legutoljára megtalált N megengedett megoldás, vagy a legutóbb megtett N lépés, illetve azok inverze. A rövidtávú memóriában tárolt elemek tabunak számítanak, tehát a megoldások nem látogathatók meg ismét, illetve a lépések nem tehetők meg újra. Az N érték a feladatban szereplő változóktól függ: $N = 7$ elégnek bizonyult tucatnyi változóval rendelkező feladatok esetén. A keresés vezérlésében fontos jellemző, hogy a rövidtávú memória felejt, azaz bizonyos számú iteráció után a legkorábban eltárolt elemet elhagyja. Ez azért lényeges, mert egy művelet általában több megengedett megoldásra is alkalmazható, és ha a keresés már kellően eltávolodott attól az s megoldástól, melyre az m műveletet alkalmazta, akkor már hasznos lehet m^{-1} alkalmazása.



11. ábra. A tabu-keresés alkalmazása kétdimenziós optimalizálási problémára

A 11. ábra a módszer működését mutatja be egy olyan kétdimenziós optimalizálási problémán keresztül, mely egy lokális és egy globális optimummal rendelkezik. Az algoritmus a kiinduló pontból feltárja a lokális minimum szomszédjait; a rövidtávú memória ekkor még nem játszik szerepet a keresés során. Azonban a lokális minimum elérése után a tabu-megszorítások miatt a keresés olyan irányba halad tovább, mely a célfüggvény értékét a legkevésbé növeli. Mivel a legutóbb meglátogatott N megoldás tabu-státusszal rendelkezik, a keresés nem haladhat tovább abba az irányba, melyből érkezett. Így az algoritmus rövid időn belül eljut a globális optimum szomszédságába.

8.2.3. Középtávú memória

A középtávú memória a keresés adott pontjáig talált M legjobb megoldást tárolja, tehát M olyan megoldást, melyeknél a célfüggvény értéke a legkisebb. Kis méretű (12 vagy kevesebb változós) problémák esetén $M = 4$ megfelelőnek bizonyult.

A középtávú memóriát a következőképpen használhatjuk fel: ha az eljárás egy meghatározott számú (kisebb problémák esetén kb. 10) lokális keresési iteráción keresztül nem talál az aktuális legjobb megoldásnál jobb megoldást, akkor a keresés intenzitását növeljük. Ez azt jelenti, hogy a jelenlegi x_{k+1} keresési lokációt egy átlagosan legjobb helyre mozgatjuk át:

$$x_{k+1} = \frac{1}{M} \sum j = 1 M y_j,$$

ahol y_j a középtávú memóriában található j. megoldáshoz tartozó változó. Ezt a lépést annak érdekében alkalmazzuk, hogy a keresést az eddig talált lejobb megoldások közelébe fókuszáljuk úgy, hogy azokat ismét ne látogassuk meg.

Annak érdekében, hogy bizonyos ígéretes területeken növeljük a keresés intenzitását, először vissza kell térnünk az eddig talált legjobb megoldások egyikéhez. Majd a tabulista méretét csökkentjük, hogy az csak kis számú iterációt engedjen meg.

Bizonyos esetekben bonyolultabb technikákat kell alkalmaznunk. Léteznek olyan optimailzálási problémák, melyek egyszerűbb alproblémákká particionálhatók. Miután megtaláltuk a partíciókon belüli optimális megoldásokat, ezek kombinációja adja a teljes probléma optimális megoldását. Ennek a módszernek a nehézsége a megfelelő partíciók kialakításában rejlik.

A számítási idő csökkentése érdekében a tabu-keresés egyes lépései során gyors heurisztikákat és ésszerű szomszédsági függvényt használunk. A keresés intenzitását növelehetjük kifinomultabb heurisztikákkal, valamint olyan szomszédsági függvénnyel, mely egy adott megoldáshoz több szomszédot rendel.

8.2.4. Hosszútávú memória

Amennyiben a keresés intenzitásának növelése nem eredményez jobb megoldást, tehát több iteráció alatt sem találunk az eddiginél jobb megoldást, érdemes a keresést eltéríteni. Ezzel elkerülhetjük, hogy az állapottér-reprezentációs gráf nagy része felfedezetlen maradjon. Az eltérítés legegyszerűbb módja, hogy az algoritmust többször újraindítjuk egy véletlen kiindulópontból. Egy másik megoldás, mely valóban garantálja a fel nem fedezett területek meglátogatását az, hogy számon tartjuk az eddig alkalmazott műveletek, illetve a feltárt megengedett megoldások bizonyos jellemzőinek gyakoriságát. Erre a célra a hosszútávú memóriát használjuk. A keresést oly módon téríthetjük el, hogy a célfüggvényben büntetjük azokat a megengedett megoldásokat, illetve műveleteket, melyek adott tulajdonságainak nagy a gyakorisága.

8.3. Algoritmus

A fentiek ismeretében a tabu-keresés algoritmusa a következő ($\tilde{f}$ a módosított célfüggvény: $\tilde{f} = f +$ Intenzitás növelése + Keresés eltérítése):

1. `Válasszuk ki az` $i \in S$ `kiinduló megengedett megoldást.`
2. `Legyen` $i^* = i$ `és` $k = 0$`.`
3. `Legyen` $k = k + 1$ `és állítsuk elő` $N(i,k)$ `egy olyan` V^* `részhalmazát, melyre a következők egyike igaz:`
   - `valamelyik` $t_r(i,m) \in T_r, (r = 1,\dots,t)$ `tabu-feltétel nem teljesül, vagy`
   - `legalább az egyik` $a_r(i,m) \in A_r(i,m), (r = 1,\dots,a)$ `aspirációs feltétel teljesül.`
4. `Válasszuk ki (`$\tilde{f}$ `figyelembe vételével) a legjobb` $j = i \oplus m \in V^*$ `megoldást`
5. `Legyen` $i = j$`.`
6. `if (`$f(i) < f(i^*)$`) then`

```
7.      Legyen i* = i.
8.  end if
9.  Frissítsük a tabu -és aspirációs feltételeket.
10. if (teljesül valamelyik megállási feltétel) then
11.     Vége.
12. else
13.     Ugorjunk a 2.  lépésre.
14. end if
```

8.4. Alkalmazások

A tabu-keresés egyik legsikeresebb gyakorlati alkalmazása az órarendkészítés. A módszer nagy sikerrel alkalmazható ütemezési problémák megoldására is. Ezeken kívül a tabu-keresés számos kombinatorikus optimalizálási probléma (pl. négyzetes hozzárendelés, utazó ügynök-probléma, egészértékű programozás) esetén használható.

Összefoglalás

Dolgozatomban a mesterséges intelligencia területén használható hatékony keresőalgoritmusokat mutattam be. Ezt a szükséges alapfogalmak ismertetésével kezdtem: megfogalmaztam, mit jelent az állapottér-reprezentáció, ezt egy konkrét példával illusztráltam. Ezután néhány alapvető gráfelméleti fogalmat vezettem be.

A második fejezetben az utazó ügynök problémájával foglalkoztam, ugyanis ezen keresztül jól szemléltethető, hogy az egyes eljárások hogyan ültethetők át a gyakorlatba egy valós probléma megoldása érdekében. Ennek a problémának is elkészítettem az állapottér-reprezentációját, majd bemutattam néhány alapvető megoldási módszert. Részletesebben kifejtettem a Lin-Kernighan algoritmust, mely speciális abból a szempontból, hogy direkt az utazó ügynök problémájának megoldására készült.

A harmadik fejezetben a keresőalgoritmusok egy osztályáról, a lokális keresőalgoritmusokról olvashatunk. A lokális keresési problémák bevezetése után a szimulált hűtés, valamint a tabu-keresés alapötleteit, működését, jellemzőit fejtettem ki. Azt is megemlítettem, hogy az utazó ügynök problémája hogyan kapcsolódik ehhez a témakörhöz.

Annak érdekében, hogy a dolgozatban szereplő módszerek közül néhánynak a működését a gyakorlatban is szemléltethessem, elkészítettem a TSP Solver nevű alkalmazást. Ennek segítségével az utazó ügynök problémáját oldhatjuk meg különböző eljárásokkal, rövid idő alatt. A felhasználói felület rajzfelületén egérkattintással hozhatunk létre egy várost, melyet egy számozott kör reprezentál. A keresés sebességének változtatásával jobban követhetjük az kiválasztott algoritmus működésének eredményét. A napló rögzíti az adott keresésre vonatkozó legfontosabb adatokat, eredményeket, melyeket akár külön szöveges fájlba is menthetünk.

A dolgozatban bemutatott módszereken kívül számos érdekes keresőalgoritmus létezik, melyeket szintén alkalmazhatunk az utazó ügynök problémájának megoldására. Például a genetikus algoritmusok érdekessége, hogy a természetben végbemenő evolúciós folyamatok mintájára működnek, a rajintelligencia-módszerek pedig a rajokban élő élőlények viselkedését veszik alapul. Ezen módszerekre összetettségük miatt a dolgozat keretein belül részletesen nem tértem ki.

Remélem, hogy munkám áttanulmányozásával érthetővé válnak a bemutatott módszerek, és nagyobb rálátást nyerünk a mesterséges intelligencia tudományára.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani konzulensemnek, Dr. Várterész Magda egyetemi docensnek a dolgozat megírásához nyújtott hasznos tanácsaiért, támogatásáért. Valamint szeretném megköszönni édesapámnak, hogy a dolgozat elkészítése után elolvasta azt, és javaslataival segítette munkámat.

Hivatkozások

[1] Alain Hertz, Eric Taillard, Dominique de Werra, A tutorial on tabu search. *EPFL, Département de Mathématiques, MA-Ecublens, CH-1015 Lausanne*.

[2] Barath Raghavan, Minimum Spanning Trees, Disjoint Sets. *UC Berkeley*, (2002).

[3] Christian Nilsson, Heuristics for the Traveling Salesman Problem. *Linköping University*.

[4] Emile H. L. Aarts, Jan H. M. Korst, Simulated annealing. *Philips Research Laboratories, Eindhoven*, (1997).

[5] Franco Busetti, Simulated annealing overview.

[6] Futó Iván, *Mesterséges intelligencia*, Aula Kiadó, Budapest, 1999.

[7] Huang, M.D., F. Romeo, A. Sangiovanni-Vincentelli, An Efficient General Cooling Schedule for Simulated Annealing. *Proc. IEEE Int. Conf. Computer Aided Design*, (1986), 381-384.

[8] Keld Helsgaun, An Effective Implementation of K-opt Moves for the Lin-Kernighan TSP Heuristic. *Department of Computer Science, Roskilde University*, (2006), 1–14.

[9] Keld Helsgaun, An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. *Department of Computer Science, Roskilde University*, (2006), 1–38.

[10] Kirkpatrick, S., Gerlatt, C. D. Jr., and Vecchi, M.P., Optimization by Simulated Annealing. *Science* **220**, (1983), 671–680.

[11] Luis Goddyn, TSP Lower Bounds. *Math*, **408**.

[12] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines *J. Chem. Phys.***21**, (1953), 1087-1092.

[13] Olivier de Weck, Cyrus Jilla, Simulated annealing: a basic introduction. *Massachusetts Institute of Technology*.

[14] S. Lin, B. W. Kernighan, An Effective Heuristic Algorithm for the Traveling-Salesman Problem, *Oper. Res.***21**, (1973), 498-516.

[15] Vanderbilt, D., S.G. Louie, A Monte Carlo Simulated Annealing Approach to Optimization over Continuous Variables. *J. Comput. Phys.* **56**, (1984), 259-271.

[16] Várterész Magda, Mesterséges intelligencia 1 előadások. *Debreceni Egyetem, Informatikai kar*, (2006), 4–9.

[17] White, S.R., oncepts of Scale in Simulated Annealing. *Proc. IEEE Int. Conf. Computer Design*, (1984), 646-651.

[18] The Metropolis Algorithm. Statistical Systems and Simulated Annealing. *Physics*, **170**.

[19] http://en.wikipedia.org/wiki/Simulated_annealing, (2009).

[20] http://en.wikipedia.org/wiki/Travelling_salesman_problem, (2009).