SZAKDOLGOZAT

Boros László

Debrecen
2008

Debreceni Egyetem

Informatika Kar

HATÉKONY KERESŐALGORITMUSOK A MESTERSÉGES INTELLIGENCIÁBAN

Témavezető:
Dr. Várterész Magda
egyetemi docens

Készítette:
Boros László
Programozó matematikus

Debrecen
2008

Tartalomjegyzék

Bevezetés

A **mesterséges intelligencia** (MI) szóval a számítástudomány egy rendkívül nagy, jelentős és nehéz (kutatási) területét szokták megjelölni. A mesterséges intelligencia kifejezés egy McCarthy nevű embertől származik, aki egy 1956-os konferencián használta először. Egyértelmű definíció a mai napig hiányzik.
Néhány ismert definíció a sok közül:

1. Az MI a mentális képességek tanulmányozása számítógépes modellek segítségével. (Charmiak, 1985)
2. Az MI az érzékelést, gondolkodást és cselekvést lehetővé tevő számítások tanulmányozása. *(Winston, 1992)*
3. Az MI olyan funkciók megvalósítására alkalmas gépek megalkotásának tudománya, mely funkciókhoz intelligenciára van szükség, amennyiben azokat emberek valósítják meg. *(Kurzweil, 1990)*
4. Az MI a számítástudomány azon ága, mely az intelligens viselkedés automatizálásával foglalkozik. *(Luger, 1993)*

Az MI kutatások célja olyan feladatok számítógépes megoldása, amelyeket nem tudunk képlettel megoldani. Ezen feladatok megoldásában az ember az intelligenciáját használja.

Jelen dolgozatomban az MI tudományterületen belül a kereső algoritmusokkal foglalkozok, melyek gráfon dolgozva valamilyen bejárási stratégiával próbálnak megoldani egy adott problémát. Bemutatok néhány fejlett kereső algoritmust, melyek közös tulajdonsága, hogy már ismert algoritmusok ötvözésével kapjuk őket hatékonyság növelése céljából. A dolgozatban főleg lokális kereső algoritmusok fognak szerepelni, ezen belül a lokális keresés alapalgoritmusa, a tabu-keresés és a szimulált hűtés. Definiálom a kombinatorikus optimalizálási probléma, a szomszédsági- függvény és gráf fogalmát. Ezek mellett bemutatom az A* algoritmus továbbfejlesztett változatát, az IDA* algoritmust is. Az algoritmusokat elemezni fogom teljesség *(completeness)*, optimalitás *(optimality)*, tárigény

(space complexity) és időigény *(time complexity)* szempontjából, majd össze is hasonlítom őket ezek alapján, amiből megtudhatjuk erősségeiket és gyengéiket. Az algoritmusokat Java nyelven is implementáltam, melyeket a „legrövidebb út” problémán teszteltem. A feladat szövege, a forráskódok és a futási eredmények a függelék részben, értékelésük az összefoglalás részben található.

Az IDA* algoritmus

(Iterative Deepening A)*

Az A* algoritmus memóriaigénye a reprezentációs gráf feltárt csúcsainak és éleinek számával arányos. Előfordulhat, hogy ez túl nagy a rendelkezésre álló memóriakapacitáshoz képest. Az iteratív mélyítés egy hasznos technika a memóriaigény csökkentésére, amit az A* algoritmus esetén is alkalmazhatunk. Az így kapott algoritmust nevezzük IDA* (iteratívan mélyülő A*) algoritmusnak. A hagyományos iteratívan mélyülő algoritmussal ellentétben itt nem mélységkorlát, hanem költségkorlát segítségével fogunk vágni a reprezentációs gráfban. Tehát az algoritmus minden iterációban minden olyan csomópontot kiterjeszt, melynek a költsége kisebb vagy egyenlő, mint az adott költségkorlát, majd a keresés aktuális iterációbeli befejeztével, ha nincs célcsúcs, a költségkorlát kitolódik (nő az értéke), és a keresés megismétlődik a gráf gyökerétől kezdve az új költséghatárral. Az új költséghatár minden egyes iteráció során meghatározódik, ami azt jelenti, hogy a legfeljebb *kh* költségű csúcsok bejárása során meg kell határozni egy új *kh'*>*kh* költséget. Ez lesz az új költséghatár.

A fentiek alapján az algoritmus vázlata a következőképpen néz ki:

```
 1: procedure IDA*(<A, kezdő, C, O>, k, h)
 2:   INICIALIZAL(<A, kezdő, C, O>, h, nyíltak)
 3:   kh ← Ø + h(kezdő)
 4:   kh' ← ∞
 5:   while IGAZ do
 6:       if nyíltak = Ø then
 7:           break
 8:       end if
 9:       csomópont ← POP(nyíltak)
10:       if ÁLLAPOT[csomópont] ∈ C then
11:           break
12:       end if
13:       if (ÚTKÖLTSÉG[csomópont]+ HEURISZTIKA[csomópont]) ≤ kh then
14:           KITERJESZT(<A, kezdő, C, O>, k, h, csomópont, nyíltak)
15:       else
16:           kh' ← MIN(kh', ÚTKÖLTSÉG[csomópont] + HEURISZTIKA[csomópont])
17:       end if
18:       if nyíltak = Ø then
19:           if kh' = ∞ then
```

```
20:                     break
21:             end if
22:             kh ← kh'
23:             kh' ← ∞
24:             INICIALIZAL(<A, kezdő, C, O>, h, nyíltak)
25:         end if
26:     end while
27:     if ÁLLAPOT[csomópont] ∈ C then
28:         MEGOLDÁS-KIÍR(csomópont)
29:     else
30:         print „Nincs megoldás"
31:     end if
32: end procedure

33: procedure INICIALIZAL(<A, kezdő, C, O>, h, nyíltak)
34:         ÁLLAPOT[új-csomópont] ← kezdő
35:         SZÜLŐ[új-csomópont] ← NIL
36:         OPERÁTOR[új-csomópont] ← *
37:         ÚTKÖLTSÉG[új-csomópont] ← Ø
38:         HEURISZTIKA[új-csomópont] ← h(kezdő)
39:         PUSH(új-csomópont,nyíltak)
40: end procedure

41: procedure KITERJESZT(<A, kezdő, C, O>, k, h, csomópont, nyíltak)
42:     for all o ∈ O do
43:             if ELŐFELTÉTEL(ÁLLAPOT[csomópont, o]) then
44:                     állapot ← ALKALMAZ(ÁLLAPOT[csomópont, o])
45:                     ÁLLAPOT[új-csomópont] ← állapot
46:                     SZÜLŐ[új-csomópont] ← csomópont
47:                     OPERÁTOR[új-csomópont] ← o
48:                     HEURISZTIKA[új-csomópont] ← h(állapot)
49:                     ÚTKÖLTSÉG[új-csomópont] ← ÚTKÖLTSÉG[csomópont] +
50:                     k(o,ÁLLAPOT[csomópont])
51:                     PUSH(új-csomópont,nyíltak)
52:             end if
53:     end for
54: end procedure
```

Az algoritmusban használt függvény- és változónevek jelentése

A *k()* egy függvény, mely meghatározza, hogy az adott csomópontba annak szülőjétől milyen költséggel jutottunk el, vagyis az ehhez szükséges operátor alkalmazásának költsége. A *h()* függvény heurisztika, ami megbecsüli az adott csomóponttól a célig vezető út költségét. Az algoritmusban szereplő *kh* változó lesz az aktuális költséghatár, melynek értéke megegyezik az aktuális csomópontig megtett út költsége és az aktuális csomóponttól a célig vezető út becsült költségének összegével (útköltség+heurisztika). A *kh'* változó értéke az aktuális iteráció során határozódik meg. Ennek értéke lesz a következő iteráció költséghatára. A *nyíltak* egy verem adatszerkezet, melybe kiterjesztés során kerülnek a csomópontok. Két művelete van: a POP() és a PUSH(). Előbbivel a tetejéről tudunk egy elemet levenni, utóbbival a tetejére egy elemet rátenni. Minden csúcsnak pontosan egy szülője lesz nyilvántartva a keresőgráfban. Egy csomópont és szülője közötti élt visszamutató segítségével realizáljuk. Megoldás találása esetén így tudjuk majd rekonstruálni a keresett utat, hogy a célcsúcstól kezdve visszamutatók segítségével a szülő csomópontok felé haladunk addig, amíg el nem jutunk a keresőgráf gyökeréig. Az INICIALIZAL() eljárás segítségével a kezdőállapotot állítjuk be, míg a KITERJESZT() eljárás segítségével az aktuális csomópontot terjesztjük ki, vagyis létrehozzuk annak gyermekeit.

Az algoritmus lépésenkénti magyarázata

Az 1. pontban található $<A, kezdő, C, O>$ négyes az adott feladat állapottér-reprezentációja, ahol

- $A \neq \emptyset$ halmaz a probléma állapottere,
- $kezdő \in A$ a kezdőállapot,
- $C \subseteq A$ a céllállapotok halmaza,
- $O \neq \emptyset$ az operátorok véges halmaza.

A 2. pontban meghívjuk az INICIALIZAL() eljárást, melynek törzse a 33-40. pontban található. Ez fogja beállítani a kezdőállapotot: a szülő csomópontra mutató mutató NIL értéket kap (ez jelzi, hogy ez lesz a gyökér csomópont), az útköltséget nullára állítja, a *h()* függvénnyel meghatározza a heurisztikát, végül a kezdőállapotot beleteszi a *nyíltak* verembe.

A 3-4. pontban megadjuk a költséghatárok kezdőértékeit. Az 5. pontban indítunk egy végtelen ciklust, mely a 26. pontig tart. A program akkor fog kilépni a ciklusból, ha célállapotot talált, vagy ha teljes egészében bejárta a reprezentációs gráfot. A 6. pontban található feltétel akkor lesz igaz, ha a gráfnak egyetlen csúcsa sincs. Ekkor kilép a program a ciklusból, és a 27. pontban folytatódik. A 9. pontban leveszünk egy elemet a *nyíltak* verem tetejéről. Ez lesz az aktuális *csomópont*. A 10. pontban található feltétel akkor teljesül, ha az aktuális *csomópont* célállapot. Ha az célállapot, a program kilép a ciklusból, és a 27. pontban folytatódik. A 13. pontban megvizsgáljuk, hogy a *csomópont* költsége (útköltség+heurisztika) nem nagyobb-e a *kh* aktuális költséghatárnál. Ha a feltétel igaz, meghívjuk a következő pontban a KITERJESZT() eljárást, melynek törzse a 41-54. pontban található. Ez az aktuális *csomópont* gyermekeit fogja bele tenni egymás után a *nyíltak* verembe, miközben beállítja azok tulajdonságait: visszamutató a szülőre, költség és heurisztika meghatározása, operátor tárolása. Ha a feltétel hamis, az aktuális csomópont költsége lesz az új költséghatár, ha az kisebb annál. A 18. pontban található feltétel teljesülése azt jelenti, hogy az adott költséghatárig bejártunk minden csomópontot. Itt két dolog lehetséges, amit a 19. pontban szereplő beágyazott feltétel dönt el. Ha ez a feltétel is teljesül, akkor a keresés nem folytatható, ami azt jelenti, hogy bejártuk a gráfot teljes egészében. Ha nem teljesül a feltétel, akkor a költséghatár kitolódik, és ismét beállítódik a kezdőállapot, vagyis a keresés újraindul a kezdőállapotból, de már az új költséghatárral. A 27-31. pontban található rész, kiírja a megoldást, ha van. Ellenkező esetben a „Nincs megoldás” kerül kiírásra.

Optimalitás és teljesség

Legyen $f(n) = k(n) + h(n)$, ahol

- n az aktuális csomópont,
- $k(n)$ az aktuális csomópontig megtett út költsége,
- $h(n)$ pedig az aktuális csomóponttól a célcsúcsig vezető út becsült költsége.

Az IDA* algoritmus ezt az $f(n)$ költséget hasonlítja össze az *aktuális költséghatárral* futás közben.

Ha *fában* keresünk, az IDA* **optimális**, *ha a heurisztika elfogadható* ***(admissible heuristic)****, vagyis soha nem becsüli felül a cél eléréséhez szükséges költséget*. Elfogadható heurisztikus függvényre legjobb példa az „utazó ügynök” probléma során alkalmazott

heurisztika, mivel bármely két pont között a legrövidebb távolság a légvonalban mért távolság, így a légvonalbeli táv soha nem becsülhet túl.

Állítás: *A fa-keresést használó IDA* algoritmus optimális, ha h(n) elfogadható.*

Bizonyítás:

Tegyük fel, hogy az utolsó kiterjesztés során megjelenik a fa peremén egy szuboptimális *G* célcsúcs, melyre *h(G)= Ø.* A **fa pereme (fringe)** egy olyan lista, amely a kifejtésre váró **levélcsomópontokat (leaf node)** tartalmaz. Az optimális megoldás költségét jelöljük *B*-vel. Ekkor:

$f(G) = k(G) + h(G) = k(G) > B.$

Vegyünk egy perembeli *n* csomópontot, mely az optimális megoldás útvonalán fekszik. Ha *h(n)* nem becsüli túl az optimális megoldáshoz vezető út folytatását, akkor

$f(n) = k(n) + h(n) \leq B.$

Ez azt jelenti, hogy $f(n) \leq B < f(G)$, így a *G* célcsúcs nem kerül kifejtésre.

Amennyiben *gráfban* keresünk, a fenti bizonyítás nem működik, mivel az algoritmus vissza térhet szuboptimális megoldással. Ez azt jelenti, hogy ha a célcsúcsba két él vezet, akkor az algoritmus elvetheti az optimálist, ha az nem elsőnek került kiszámításra. E probléma kétféle módon oldható meg. Az egyik, hogy a gráf-kereső algoritmust kiegészítjük úgy, hogy a csomópontba vezető két út közül a drágábbat vesse el. A másik megoldás során biztosítani kell a *h(n)* heurisztikus függvény **monotonitását (monotonicity)**. Egy heurisztikus függvény akkor és csak akkor monoton, ha teljesíti a **háromszög-egyenlőtlenséget**. A háromszög-egyenlőtlenség kimondja, hogy egy háromszög bármely két oldalának összege nem lehet kisebb a harmadik oldalánál. A légvonalban mért távolság teljesíti a háromszög-egyenlőtlenséget, így az monoton.

Állítás: *A gráf-keresést használó IDA* algoritmus optimális, ha h(n) monoton.*

Állítás: *Ha h(n) monoton, akkor f(n) értékek akármilyen út mentén nem csökkennek.*

Bizonyítás:

Legyen *n* egy csomópont, *n'* pedig egy utódja. Jelöljük *c(n,n')*-vel az *n*-ből *n'*-be vezető út költségét. Ekkor:

$k(n') = g(n) + c(n,n')$

és

$f(n') = k(n') + h(n') = k(n) + c(n,n') + h(n') \geq k(n) + h(n) = f(n).$

Az IDA* ugyanolyan feltételek mellett **optimális** és **teljes**, mint az A*, de a mélységi keresés jellege miatt csak a leghosszabb felderített út hosszával arányos memóriát igényel.

Tárigény és időigény

Az algoritmus memóriaigénye lineáris a keresés során feltárt leghosszabb út hosszával. Jelöljük *b*-vel, hogy a reprezentációs gráfban egy csomópont átlagosan hány csomópontot terjeszt ki (elágazási tényező), *d*-vel pedig a célcsúcs gráfbeli mélységét.
Ekkor a tárigény **O(b*d)**, az idő igény **$O(b^d)$.**

Vizsgáljuk meg részletesebben az algoritmus időigényét, amennyiben egy célcsúcs elérhető a kezdőállapotból!

A következőket feltételezzük:

- az állapottér faszerkezetű
- vezessük be az *f()* függvényt (útköltség+heurisztika), mint költséget, mely monoton növekvő az utak mentén, vagyis ha az *n* csomópontból az *m* csomópont közvetlenül elérhető, akkor teljesül az $f(n) \leq f(m)$.
- legyen V_k a *k*-adik legkisebb *f()* értékű csúcsok halmaza. Tehát az ebbe a halmazba tartozó csúcsok költségei rendre megegyeznek. Minden $n,m \in V_k$ esetén

- $f(n) = f(m)$ és minden $j<k$ esetén $V_j<V_k$. Ez azt jelenti, hogy minden V_j-beli csomópont költsége kisebb lesz bármely V_k-beli csomópont költségénél.
- Mivel a $f()$ függvény monoton növekvő, ebből következik, hogy a kezdőcsúcs a legkisebb indexű, V_0 halmazban fog szerepelni, továbbá, ha $n \in V_j$, akkor n gyermekei csak azokba a V_k halmazokba eshetnek, melyekre $j \leq k$.
- legyen b egy heurisztikus elágazási tényező, melyre $b = |V_{j+1}| / |V_j|$ minden $j \geq 0$ esetén. Ekkor $|V_k| = b^k * |V_0|$
- $b > 1$ és $|V_0| = 1$

Az időigény kiszámítása példán keresztül

Az algoritmus először a V_0 halmazbeli csúcsokat terjeszti ki. Ezután újraindul a mélységi keresés és kiterjeszti a V_0 és V_1 halmazokba eső csúcsokat. Ha a V_d az első olyan halmaz, mely célcsúcsot tartalmaz, akkor a mélységi keresés összesen d alkalommal indul újra.

Tehát $N_0+N_1+\ldots+N_d$ csúcsot terjeszt ki összesen, ahol

$N_0 = O(1)$,

$N_1 = O(1+b)$,

$N_2 = O(1+b+b^2)$,

$N_d = O(1+b+b^2+\ldots+b^d)$.

Legyen b=3 és d=4.

Ekkor egy egyszerű keresőgráf csomópontjainak száma: 1+3+9+27+81=121

A kiterjesztés a legmélyebb szint csomópontjait egyszer hozza létre, az eggyel magasabban lévőket kétszer, a kettővel magasabban lévőket háromszor és így tovább.

Az így kiterjesztett csomópontok száma: 5+12+27+54+81=179

Másképpen, a fenti képletek segítségével:

$N_0 = O(1) = 1$

$N_1 = O(1+b) = 1+3 = 4$

$N_2 = O(1+b+b^2) = 1+3+9 = 13$

$N_3 = O(1+b+b^2+b^3) = 1+3+9+27 = 40$

$N_4 = O(1+b+b^2+b^3+b^4) = 1+3+9+27+81 = 121$

$N_0+N_1+N_2+N_3+N_4 = 1+4+13+40+121 = 179$

Az A* algoritmussal összehasonlítva $b>1$ esetén az IDA* nagyjából azonos futási idejű (exponenciálisan növekvő), mint az A*, de sokkal kevesebb memóriát használ. Az A* memóriaigénye a kiterjesztett csomópontokkal és a feltárt gráf éleinek számával arányos. A rendelkezésre álló memóriakapacitáshoz képest ez gyakran túl nagy lehet, vagyis az algoritmus hamar teleírhatja a memóriát anélkül, hogy megtalálná az optimális célállapotot.

Lokális kereső algoritmusok

A lokális kereső algoritmusokat **közelítő algoritmusoknak** *(approximation algorithm)* nevezzük, melyek nem garantálják az optimális megoldás megtalálását. Megoldásról megoldásra lépnek és céljuk egy *lokálisan legjobb* megoldás megtalálása. A célhoz vezető utat nem tárolják, csak az aktuális állapotot, ahonnan egy szomszédos állapotra lépnek tovább.

A kombinatorikus optimalizálási probléma

Adott egy kombinatorikus optimalizálási probléma, amit P-vel jelölünk. P-nek x egy példánya, mely meghatározza a megengedett megoldások halmazát, valamint az $f_x()$ optimalizálandó célfüggvényt. A megengedett megoldások halmazát jelöljük $S_i(x)$-el. Keresünk egy s megengedett megoldást, melyre teljesül, hogy minden $r \in S_i(x)$ megengedett megoldásra $f_x(r) \geq f_x(s)$ teljesül minimalizálási, illetve $f_x(r) \leq f_x(s)$ maximalizálási probléma esetén. Azokat az s megengedett megoldásokat, melyek teljesítik az előbbi feltételt, **optimális** vagy **globálisan optimális megoldásoknak** nevezzük.

A szomszédsági függvény

A megengedett megoldások $S_i(x)$ halmazán definiáljunk egy **szomszédsági függvényt**, ami minden megengedett megoldáshoz hozzárendeli $S_i(x)$ egy részhalmazát, mely olyan megengedett megoldásokat tartalmaz, melyek közelinek tekinthetők. Jelöljük a szomszédsági függvényt N_i-vel. Ekkor s tetszőleges megengedett megoldás szomszédait az $N_i(s,x) \subseteq S_i(x)$ megengedett megoldások alkotják.

Az N_i szomszédsági függvényt **transzformáicós szabályok** segítségével definiáljuk, amiket megengedett megoldások módosítására, szomszédos megengedett megoldások előállítására használunk. Lokális optimalizálási feladatot úgy kapunk, hogy definiáljuk a transzformációs szabályok egy M halmazát. Az M elemeit megengedett megoldások módosítására használjuk. Ha $M_i(s)$ az s megengedett megoldásra alkalmazható

transzformációs szabályok halmaza, akkor $N_i(s,x) = \{ r \in S_i(x) \mid \exists m \in M_i(s) : r=m(s) \}$, ahol m egy M halmazbeli transzformációs szabály.

A lokális keresési probléma

A lokális keresési probléma során keresünk egy olyan $s \in S_i(x)$ megengedett megoldást, mely **lokálisan optimális**, vagyis minden r $\in N_i(s,x)$ megengedett megoldásra $f_x(r) \geq f_x(s)$ teljesül minimalizálási, illetve $f_x(r) \leq f_x(s)$ maximalizálási probléma esetén.
Ha egy megengedett megoldás globálisan optimális, akkor lokálisan is optimális. Fordítva ez nem mindig igaz.

Egy adott optimalizációs feladathoz különböző szomszédsági függvényeket definiálhatunk, ezáltal különböző lokális keresési problémákat kaphatunk ugyanazon keresési problémából. Ha minden lokálisan legjobb megoldás egyben globálisan is a legjobb, akkor a **szomszédsági függvény exakt**.

A szomszédsági gráf

A **szomszédsági gráf** egy szomszédsági függvény által kifeszített irányított gráf, melynek csúcsai megengedett megoldások.

Tulajdonságai:

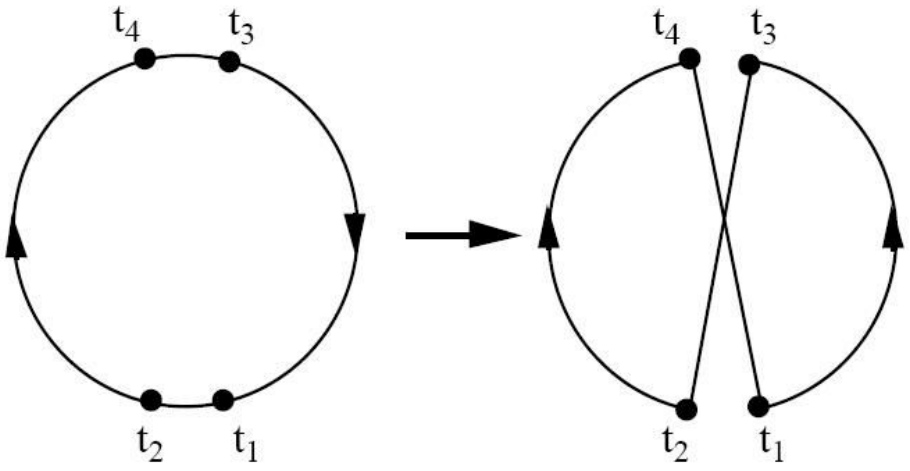
- Csúcsai megengedett megoldások.
- Ha s és r megengedett megoldások, akkor irányított él vezet s-ből r-be, ha $r \in N(s,x)$.
- Egy vagy több maximális komponensből áll, ahol két csúcs ugyanahhoz a komponenshez tartozik, ha a két csúcs között irányított út van oda-vissza.
- Egy adott csúcsból nem biztos, hogy elérhető minden más csúcs irányított élsorozat mentén.

A 2-OPT és a 3-OPT szomszédsági függvények

Oldjuk meg lokális kereséssel az *„utazó ügynök”* problémát. Adott *n* darab város. Keressük a legrövidebb bejárást úgy, hogy közben minden várost érintenünk kell pontosan egyszer. A kereséshez használt gráf teljes, ami azt jelenti, hogy minden városból minden város elérhető egyetlen él mentén. Két tetszőleges *a* és *b* város távolságát jelöljük *d(a,b)*-vel. A megengedett megoldások halmazát a bejárási sorrendek alkotják, ami jelen esetben az *1,2,3,…,n* számok összes permutációja. Egy olyan *P* permutációt keresünk, melyre $d(P_1,P_2)+ d(P_2,P_3)+\ldots+ d(P_n,P_1)$ úthossz a legkisebb. Ekkor *(n-1)!* bejárás lehetséges, ami nagyon sok, és a legjobb megengedett megoldás megtalálásához az összes permutációt meg kellene vizsgálnunk.

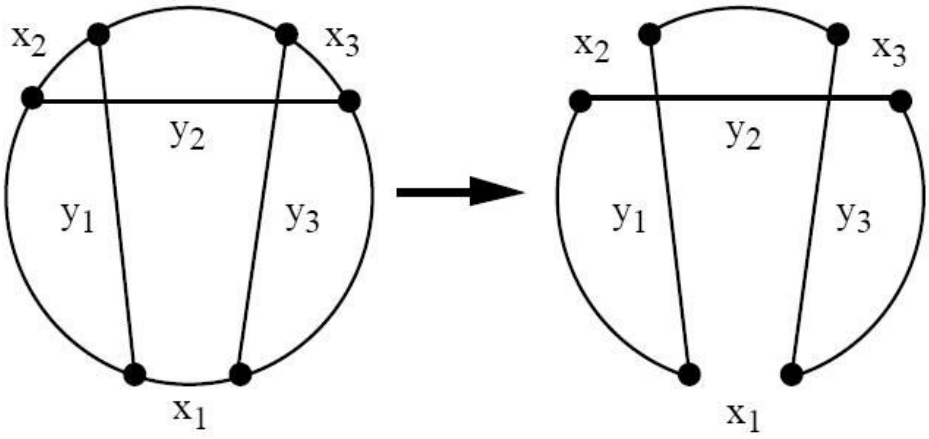
Ehelyett megpróbáljuk közelíteni a megoldást. Ehhez szomszédsági függvény segítségével lokális keresési problémát definiálunk. A szomszédsági függvény az összes megengedett megoldás halmazát leszűkíti annak egy részhalmazára. Ebből következik, hogy ha az optimális megengedett megoldás e részhalmazon kívül van, akkor azt nem találjuk meg. Különböző szomszédsági függvényekkel különböző részhalmazokat kaphatunk.

Két ismert szomszédsági függvény a *2-OPT* és a *3-OPT*. Egy adott bejárási sorrend (kör) 2-OPT szomszédját úgy kapjuk, hogy választunk két nem szomszédos élet a bejárásban, ezeket töröljük, és a törölt élekhez tartozó csúcsokat keresztbe kötjük.



Az ábrán kiindulunk egy $t_4\ t_3\ t_1\ t_2$ bejárási irányú körből. Ezt a kört transzformáljuk a *2-OPT* szomszédsági függvénnyel, így a bejárás megváltozik $t_4\ t_1\ t_3\ t_2$-re. A transzformáció akkor eredményes, ha $d(t_4,t_3)+d(t_1,t_2) > d(t_4,t_1)+d(t_3,t_2)$.

A *3-OPT* hasonló a *2-OPT*-hoz, de itt már 3 élet törlünk, és több lehetőség van az útrészletek egyesítésére.



A fenti ábrán az első rajz egy egyszerű kör, melyen a bejárás iránya az óramutató járásával azonos. A *3-OPT* szomszédsági függvény kiválaszt három nem szomszédos élet, törli azokat, majd a törölt élekhez tartozó csomópontokat keresztbe köti az ábrán látható módon, ezáltal egy új bejárási irányt kaptunk.

Tehát az „utazó ügynök” probléma esetén előállítunk egy permutációt, majd arra addig alkalmazzuk a *2-OPT* vagy *3-OPT* szomszédsági függvény valamelyikét, míg egy lokálisan legjobb megoldást nem kapunk.

A lokális keresés alapalgoritmusa

Adott egy kombinatorikus optimalizálási probléma, és egy szomszédsági függvény a megengedett megoldásokon definiálva, mellyel lokális keresési problémát kapunk.

Cél: egy lokálisan optimális megengedett megoldás találása a megengedett megoldások halmazán.

A keresés egy ciklus, ami mindig a javuló értékek felé halad. Az algoritmus egy szomszédsági gráfon fog dolgozni. A szomszédsági gráf minden csomópontja megengedett megoldás, amiből a lokálisan optimálist fogjuk megkeresni.

Az algoritmus lépései:

1. Válasszunk egy tetszőleges csomópontot a szomszédsági gráfból, és legyen ez az *aktuális* megengedett megoldás.
2. Keressünk egy olyan megengedett megoldást az *aktuális* megengedett megoldás szomszédságában, amely az *aktuális* megengedett megoldásnál jobb.
   - ha találunk egy ilyen szomszédot, akkor az lesz az új *aktuális* megengedett megoldás, és a 2. pontra ugrunk.
   - ha nem találunk ilyen szomszédot, akkor megállunk, és az *aktuális* megengedett megoldás lesz a lokálisan optimális.

```
1: procedure Lokális-keresés-alapalgoritmusa(<A, kezdő, C, O>, k)
2:   aktuális ← kezdő
3:   while IGAZ do
4:     O' ← { o | o ∈ O ∧ ELŐFELTÉTEL(aktuális,o) ∧
5:           ∧ k(ALKALMAZ(aktuális,o)) ≤ k(aktuális) }
6:     if O' ≠ Ø then
7:           operátor ← VÁLASZT({ o | o ∈ O' ∧
8:     ∧ ∀o'( o' ∈ O' → k(ALKALMAZ(aktuális,o)) ≤ k(ALKALMAZ(aktuális,o')))})
9:           aktuális ← ALKALMAZ(aktuális,operátor)
10:    else
```

```
11:              break
12:     end if
13:   end while
14:   print aktuális
15:   end procedure
```

Az algoritmus magyarázata

Első lépésben választunk egy kezdeti megengedett megoldást (2. pont). Innen fog indulni a keresés. A 3. pontban indítunk egy végtelen ciklust. Maga az algoritmus egyetlen ciklus, ami mindig a javuló értékek felé halad, vagyis az *aktuális* megengedett megoldásnál jobb megengedett megoldást keres a szomszédsági függvény által hozzá rendelt megengedett megoldások halmazában, majd ezt az új állapotot veszi fel aktuálisként, feltéve, hogy létezik ilyen. Ha nincs az aktuálisnál job megengedett megoldás, akkor a program kilép a ciklusból . Erről gondoskodik a 6. pontban található feltétel **else** ága. A 4. pontban az *O'* halmazba felvesszük azon operátorokat, melyek alkalmazhatóak az *aktuális* állapotra. Itt ügyelünk arra, hogy olyan operátor ne kerüljön a halmazba, melyet alkalmazva az *aktuális* állapotra, annál rosszabb megoldáshoz jutunk. A *k()* függvény (célfüggvény) a megengedett megoldások költségét határozza meg. E költség-függvény segítségével hasonlítjuk össze a megengedett megoldásokat. Mindig a legkisebb költségű irányba fogunk továbblépni. A 6. pontban megvizsgáljuk, hogy van-e eleme az *O'* halmaznak. Ha nincs, akkor kiugrunk a ciklusból (11. pont), a program pedig a 12. pontban folytatja futását, ahol kiírja a megoldást. Ha teljesül a 6. pontban található feltétel, akkor kiválasztjuk azt az operátort, amely a legjobb megoldásba visz tovább, majd alkalmazzuk az *aktuális* állapotra.

Hogyan befolyásolja a szomszédsági függvény megválasztása az algoritmus hatékonyságát?

Jelöljük *x*-el a kombinatorikus optimalizálási feladat egy példányát, amely meghatározza a megengedett megoldások *S(x)* halmazát. Legyen *s* egy megengedett megoldás. Tegyük fel, hogy van két szomszédsági függvényünk, *SZF1(s,x)* és *SZF2(s,x)*, azzal a tulajdonsággal,

hogy *SZF1(s,x)* részhalmaza *SZF2(s,x)*-nek, ami azt jelenti, hogy *SZF2(s,x)* szomszédsága bővebb.

- *SZF2()*-ben lehet olyan *r* megoldás, amely *SZF1()*-ben nem szerepel, és *r* jobb megengedett megoldás, mint bármely *SZF1()*-beli.
- Mivel *SZF2()* szomszédsága bővebb, ezért több időre és tárra van szükség az abban való keresésre, mint *SZF1()* esetében.

Azonos számú iteráció esetén *SZF2()*-vel az algoritmus erőforrás-igényesebb, mivel előfordulhat, hogy egy-egy iterációban több lépést kell megtenni, mint *SZF1()*-el.

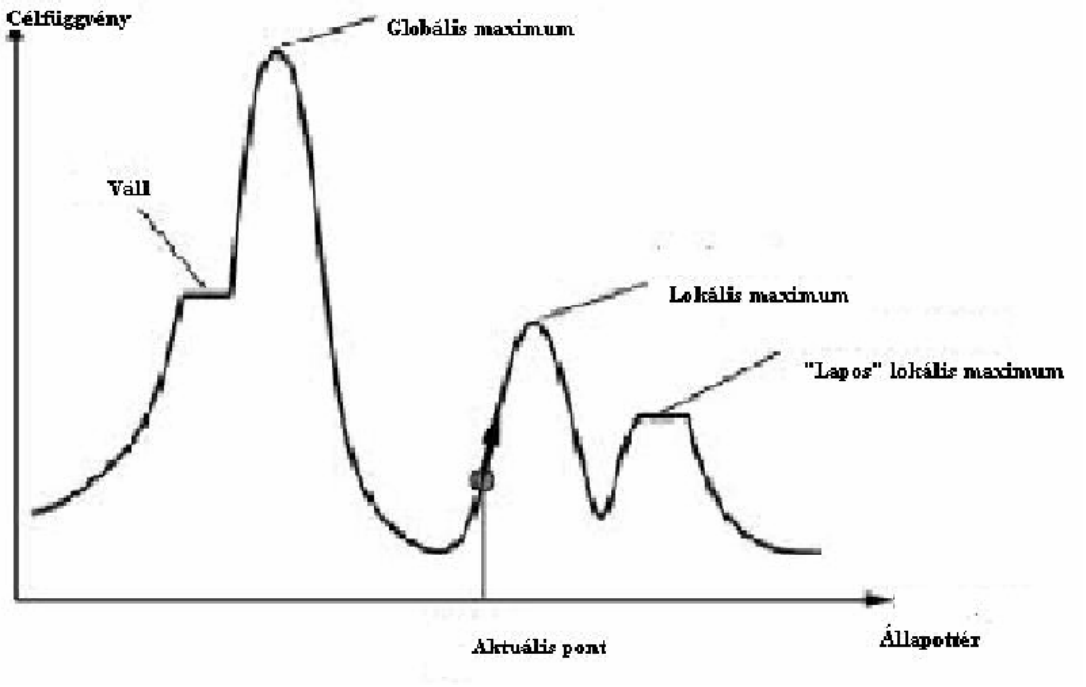
Kérdés: sok iterációt tegyünk meg gyorsan, vagy kevesebbet, de egy-egy iterációra több időt szánva?

Teljesség

Az algoritmus nem teljes. Éppen ezért nem is garantált az optimális megoldás megtalálása.

Optimalitás

Mivel ez egy közelítő algoritmus, az optimális vagy más néven globálisan optimális megoldás nem garantálható.



Ez egy állapottérfelszín, ami a lokális keresés megértését segíti. Vannak pontjai, melyek az állapotok, és vannak "magasságai", amiket a célfüggvény határoz meg. Ha a magasság a költséggel arányos, akkor a legalacsonyabban fekvő völgyet, a **globális minimumot** keressük. Ha a magasság célfüggvénynek felel meg, akkor a **globális maximumot** keressük. Az algoritmus 5. és 8. sorában a ≤ és ≥ hasonlító operátorok megfelelő alkalmazásával adható meg, hogy minimalizálni vagy maximalizálni akarunk-e. Az ábráról leolvasható, hogy az *"aktuális pont"* lokális maximuma csak lokálisan optimális megoldást eredményez, globálisat már nem, mivel a pont a *"lokális maximum"* felé halad, és ott megáll. Ha az aktuális pont valahol az első hegyen lenne, akkor a lokálisan optimális megoldása egyben globálisan optimális megoldás is lenne.

A *fennsík-probléma* azt jelenti, hogy ha a keresés során eljutunk a *"lapos lokális maximum"*-hoz, nem biztos, hogy az algoritmus megtalálja onnan a kivezető utat.
Viszont a *váll*ról (ami szintén egy fennsík) még vezet út felfelé. Tehát, ha megengedünk *oldallépéseket*, akkor a *váll*ról tovább tudunk lépni. Igy a váll nem okozhat problémát. A *oldallépésekkel* viszont az a gond, hogy ha már nem vezet út felfelé (mint a *"lapos lokális maximum"*-nál), akkor végtelen hurokba kerülünk.
Megoldás: korlátozzuk az oldallépések számát.

Időigény

Lineárisan nő az érintett csomópontok számával. Jó helyzetből indulva gyorsan célhoz ér. Tegyük fel, hogy egy adott problémával kapcsolatban a *célfüggvény* csak véges sok különböző értéket vehet fel minden x problémapéldányon. Jelöljük ezt a korlátot $K(x)$-el, amely korlát az algoritmus lépésszámára is. Ekkor az algoritmus legfeljebb $O(K(x))$ lépés után megáll. Ha van olyan $p(z)$ polinom, hogy minden x problémapéldányra $K(x) \leq p(|x|)$ teljesül, akkor az algoritmus időbonyolultsága $O(p(|x|)$, vagyis polinomiális.

Tárigény

A tárigény kicsi, mivel keresési gráfot, utat nem tárol, csak a pillanatnyi *aktuális* állapotot. Például a 8-királynő problémánál nem utat keresünk, hanem egy állapotot, melyre teljesülnek bizonyos feltételek.

Az algoritmus előnye, hogy nagy vagy végtelen keresési térben sokszor elfogadható megoldást adnak ott, ahol a szisztematikus keresők nem tudnának.

A szimulált hűtés

(simulated annealing)

A lokális keresés alapalgoritmusának hátránya, hogy gyakran lokális optimumba ragad. A szimulált hűtés ezt a problémát oldja meg, vagyis tekinthető az előző algoritmus javításának is. Kiindulunk egy véletlenszerűen kiválasztott megoldásból, és egy új megoldást keresünk annak környezetében, viszont a *legjobb* lépés helyett egy *véletlen* lépést teszünk. Egy lépés mindig végrehajtásra kerül, ha az jobb megengedett megoldásba visz. Ellenkező esetben a lépést csak valamilyen 1-nél kisebb valószínűséggel tesszük meg. A valószínűség exponenciálisan csökken a lépés romlásának mértékével. A valószínűséghez használunk egy hőmérséklet paramétert is. A rossz lépések valószínűsége a hőmérséklet csökkenésével csökken. A hőmérséklet csökkenés mértékét a *hűtési ütemterv* határozza meg, melyet bemenetként kap meg az algoritmus. A rossz lépéseknek köszönhetően az algoritmus ki tud lépni a lokális optimumból.

Az algoritmus alapötlete a következő fizikai eljárásból ered:

Adott egy test, amit *alapállapotba* akarunk hozni, ahol az anyagi részecskék jól struktúrált kristályrácsba rendeződnek, és a test belső energiája minimális. Először felmelegítjük a testet, hogy az megolvadjon és a részecskék szabadon mozoghassanak, majd lassan lehűtjük, hogy a részecskék az alapállapot szerint elrendeződjenek. Ha az olvasztott anyag nem elég meleg, vagy a hűtést nem megfelelő ütemben végzik, akkor a kristályrácsba-rendeződés az alapállapot felvétele előtt megakad.

A szimulált hűtést egy kombinatorikus optimalizálási feladat megoldására használják.
A következőket feltételezzük:

- a feladat problémapéldányának megengedett megoldásai egy test és a testet alkotó részecskék állapotainak felelnek meg.
- egy megengedett megoldás *k()* költsége ekvivalens az állapothoz tartozó energiával.
- bevezetünk egy **hőmérséklet paramétert.**

```
1: procedure Szimulált-hűtés(<A, kezdő, C, O>, k, hűtési_ütemterv)
2:    aktuális ← kezdő
3:    következő ← NIL
4:    j ← 0
5:    T0 ← hűtési_ütemterv(j,T)
6:    L0 ← hűtési_ütemterv(j,L)
7:    while IGAZ do
8:          for i ← 1 to Lj do
9:                O' ← { o | o ∈ O ∧ ELŐFELTÉTEL(aktuális,o) }
10:               operátor ← VÁLASZT({ o | o ∈ O' })
11:               következő ← ALKALMAZ(aktuális,operátor)
12:               if k(következő) ≤ k(aktuális) then
13:                         aktuális ← következő
14:               else if exp( (k(aktuális)-k(következő)) / Tj ) > random[0,1) then
15:                         aktuális ← következő
16:                   end if
17:               end if
18:         end for
19:         j ← j+1
20:         Tj ← hűtési_ütemterv(j,Tj-1)
21:         Lj ← hűtési_ütemterv(j,Lj-1)
22:         if MEGÁLLÁSI_FELTÉTEL then
23:               break
24:         end if
25:   end while
26:   print aktuális
27:   end procedure
```

Az algoritmus magyarázata

Első lépésben választunk egy kezdeti megengedett megoldást (2. pont). Innen fog indulni a keresés. Továbbá választunk egy kezdeti hőmérsékletet (5. pont) és egy kezdő folyamathosszt (6. pont). A kezdeti folyamathossz meghatározza, hogy a kezdeti hőmérsékleten hány átmenet történjen meg. Ennyiszer fog lefutni a belső ciklus.

Az algoritmusnak van egy úgynevezett *hűtési ütemterve*, amit paraméterben kap meg. Egy későbbi fejezetben tárgyalom, hogy mi is ez. Most elég annyi, hogy ezzel adjuk meg a *T* és *L* értékeit *j* függvényében, valamint a külső ciklus *megállási feltételét*.

A belső ciklus hasonló a lokális keresés alapalgoritmusának ciklusához azzal a különbséggel, hogy itt rosszabb irányba is tovább tudunk haladni, vagyis most már nem csak előre tudunk

haladni, hanem hátra is. Először is az *aktuális* megengedett megoldás szomszédságából választunk egy tetszőleges szomszédot (9-11. pont).
Ezután megvizsgáljuk, hogy ezt a szomszédot elfogadjuk-e új *aktuális* állapotnak (12-17. pont).
Az elfogadás két egymásba ágyazott feltételből áll:

- ha a választott szomszéd költsége nem rosszabb, mint az *aktuális* megengedett megoldásé, akkor ez lesz az új *aktuális* állapot (12-13. pont).
- ha választott szomszéd költsége rosszabb, akkor generál egy véletlenszámot 0 és 1 között, és összehasonlítja az *exp((k(aktuális)-k(következő)) / Tj)* értékkel (ezt *Metropolis kritériumnak* is szokás nevezni).
    - ha a véletlenszám ettől kisebb, akkor a választott szomszéd lesz az új *aktuális* állapot, és ezzel folytatja a keresést.
    - ha nem kisebb, akkor a keresés a jelenlegi *aktuális* állapottal folytatódik.

Miután a belső ciklus befejezte a javítási kísérleteket az adott T_j hőmérséklethez, növeljük *j* értékét egyel, továbbá új hőmérésklet és folyamathossz kerül meghatározásra a *hűtési ütemterv* által a *j*, T_{j-1} és L_{j-1} értékek függvényében (19-21. pont). Ezután a belső **for** ciklus ismét elindul a megváltozott értékű paraméterekkel.
Általános szabály, hogy a **folyamathossz nem csökkenő** sorozat, míg a **hőmérséklet szigorúan csökkenő** sorozat legyen.
A hőmérséklet szerepe a belső ciklusban az, hogy a keresés folyamán (miközben a *T* nullához tart) az algoritmus egyre kisebb valószínűséggel fogadjon el az aktuálisnál rosszab megengedett megoldást. Az algoritmust a *megállási feltétellel* tudjuk befejeztetni (22-24. pont), ami tetszőleges lehet. Például megadhatjuk a *hűtési ütemtervet*, amely tartalmaz egy legalacsonyabb hőmérsékletet, és a feltétel akkor teljesül, ha *Tj* ezen érték alá esik.

Optimalitás

Az optimális megoldás megtalálása a *hűtési ütemtervtől* függ. Ha a hőmérséklet paramétert kellően lassan csökkenti, akkor az algoritmus **globális minimumot** fog találni, vagyis optimális. A hőmérséklet túl gyors csökkentése a célfüggvény lokális minimumhelyében való elakadást eredményezheti.

Időigény

Lineárisan nő az érintett csomópontok számával.

Tárigény

A tárigény kicsi, mivel keresési gráfot, utat nem tárol, csak a pillanatnyi *aktuális* állapotot.

A szimulált hűtés elemzése

Vezessük be a következő előfeltételeket és jelöléseket:

- az algoritmus „igazi" véletlenszám-generátort használ.
- legyen X_k valószínűségi változó, ahol k index (k=1,2,3,…).
- minden s megengedett megoldásra $|SZF(s,x)| = \theta$, ahol θ közös elemszám, vagyis minden s megengedett megoldás szomszédsága ugyanolyan méretű.
- T a hőmérséklet.
- $S(x)$ a megengedett megoldások halmaza
- w_{sr} jelölje a súlyokat
- $A_{sr}(T)$ az elfogadási valószínűség

Az algoritmus belső ciklusára tekintsünk úgy, mintha egy diszkrét paraméterű véletlen folyamat lenne, azaz van olyan $\{X_0, X_1, X_2,\ldots\}$valószínűségi változó sorozat, hogy az egymást követő valószínűségi változók felvett értékei az *aktuális* megengedett megoldások.
Az $\{X_0, X_1, X_2,\ldots\}$ egy **diszkrét paraméterű Markov lánc**, mivel az X_n valószínűségi változó eloszlása csak az X_{n-1} valószínűségi változó értékétől függ**.**

Az *s* megengedett megoldásról az r megengedett megoldásra való átmenet valószínűségét *T* hőmérsékleten keresztül a következő képlet adja:

- minden $s \neq r \in S(x)$ megengedett megoldásra $P_{sr}(T) = w_{sr}A_{sr}(T)$.

A w_{sr} súlyok az egyes szomszédok választásának a valószínűségét adják. Mivel minden szomszéd ugyanolyan valószínűséggel választható, ezért $w_{sr} = 1/\theta$, ha $r \in SZF(s,x)$, különben $w_{sr} = 0$ (9-11. pont).
Az elfogadási valószínűség $A_{sr}(T) = \exp(-(k(\text{következő})-k(\text{aktuális}))^{+} / T)$. (12-17. pont)
Ez azt jelenti, hogy ha $k(\text{következő})-k(\text{aktuális}) \leq 0$, akkor $\exp(-0 / T)=1$, ami annak felel meg, hogy az algoritmus mindig elfogad egy *r* szomszédot, ha az nem rosszabb, mint az *aktuális* megengedett megoldás (12-13. pont). Ha $k(\text{következő})-k(\text{aktuális}) > 0$, akkor a kifejezés az algoritmus 14. pontjának elfogadási valószínűségével ekvivalens.

Mivel az átmeneti valószínűség független az iterációk számától, ezért a Markov lánc **homogén**.

<u>*Tétel (Aarts 1989):*</u> Adott egy kombinatorikus optimalizációs probléma, és a probléma egy *x* példánya. Tegyük fel, hogy a választott *SZF()* szomszédsági függvény teljesíti a következő feltételt:

- Minden $s,r \in S(x)$ megengedett megoldásra az s-ből az *r* véges sok szomszédon belül elérhető.

Ekkor a belső ciklusban rögzített *T* hőmérséklet mellett, az X_0 értéktől függetlenül, elegendően sok iteráció megtétele után a következő stacionárius eloszlással adott annak a valószínűsége, hogy az aktuális megengedett megoldás valamely $r \in S(x)$:

$$q_r(T) = \lim_{n\to\infty} P(X_n = r \mid X_0 = s) = \exp(-k(\text{következő}) / T) / (\sum_{i\in S(x)} \exp(-k(i) / T)),$$

minden $s \in S(x)$ esetén.

A stacionárius eloszlás azt jelenti, hogy elég sok iteráció után az *aktuális* megengedett megoldás eloszlása nem változik.

A tétel szomszédsági függvénnyel kapcsolatos feltétel része ekvivalens azzal, hogy az szomszédsági gráf **erősen összefüggő** kell legyen (minden csúcsból minden csúcs elérhető irányított út mentén), ami azzal ekvivalens, hogy a Markov lánc **irreducibilis**.

A tétel következménye: A tétel feltételei mellett a szimulált hűtés algoritmusa egy optimális megoldáshoz konvergál.

Hűtési ütemtervek

Egy hűtési ütemterv a hőmérsékletparaméterre vonatkozóan a következő adatok megadását írja elő:

- T_0 kezdeti hőmérséklet.
- a hőmérséklet csökkentésének menete.
- a megállási feltételben szereplő legalacsonyabb hőmérséklet.
- a véletlenfolyamat hossza minden hőmérsékletre.

A kezdeti hőmérsékletet válasszuk olyannak, hogy erősen összefüggő szomszédsági gráf esetén minden megengedett megoldás véges sok lépésben nagy valószínűséggel elérhető legyen.

A T_j és L_j paramétert válasszuk olyannak, hogy a megengedett megoldások eloszlása jól közelítse a $q_r(T_j)$ stacionárius eloszlást L_j átmenet után.

Ha a hőmérsékletetet kicsivel csökkentjük, akkor kevesebb átmenetre van szükség a stacionárius eloszlás ismételt megközelítéséhez, mintha nagyobb lépésekben csökkentenénk. Igy viszont több hőmérsékletcsökkentésre van szükség a végső hőmérséklet eléréséhez.

Példa hűtési ütemtervre:

- A kiinduló hőmérséklet legyen egy kicsi T_0>1 érték, és ezzel teszteljük az elfogadási valószínűséget a belső ciklus lefuttatásával. Ha T_0 mellett az elfogadási arány nincs elég közel 1-hez, akkor T_0-t megszorozzuk egy egynél nagyobb számmal, majd megismételjük a tesztelést. Mindezt addig ismételjük, amíg az elfogadási arány nem kerül elég közel 1-hez.
- A hőmérséklet csökkentése egy egynél kisebb *konstanssal* való szorzással történik: $Tj = c * T_{j-1}$, ahol c egy 1-nél kisebb konstans.
- A kapott hőmérsékletsorozatra nem adunk pozitív alsó korlátot. Ehelyett a megállási feltétel akkor fog teljesülni, ha a belső ciklusban az L_j-adik iterációban talált aktuális megengedett megoldáson a célfüggvény értéke nem változik az utolsó néhány hőmérsékleten, vagyis a hőmérséklet csökkenése mellett a belső ciklus mindig ugyanahhoz a célfüggvény értékhez tart.
- Megadunk egy nagy L konstanst, amellyel az L_j sorozat határtalan növekedését korlátozzuk

Gráfszínezés szimulált hűtéssel

Adott egy $G=(V,E)$ egyszerű gráf, amely végesen sok csúcsot és élet tartalmaz. Gráf *k-színezésének* nevezzük a gráf csúcsainak egy osztályozását $V_1,\ldots,V_k$ részhalmazra, mely teljesíti az alábbi két feltételt:

- minden $1 \leq j \leq k$ esetén a V_j-beli csúcsok páronként függetlenek, vagyis nem vezet köztük él.
- minden gráfcsúcs pontosan egy részhalmaznak eleme, vagyis a $V_1, V_2, \ldots, V_k$ halmazok együttese a gráf csúcsainak egy **partíciója**.

Keresünk egy olyan részhalmazokra bontást, amely eleget tesz a fenti két feltételnek, és amely a lehető legkevesebb részhalmazból áll, vagyis *k-minimális*. Úgy is

fogalmazhatnánk, hogy egy *G* gráf k-színezhető, ha bármely két éllel összekötött csúcsa különböző színű. Bármely fa gráf csúcsai két színnel jól színezhetők.

Definíció: A legkisebb k számot, melyre létezik színezés k színnel, a G gráf ***kromatikus számának*** *nevezzük, és χ(G)-vel jelöljük.*

Tétel: Egy végesen sok csúcsot és élet számláló G gráf esetén $\chi(G) \leq \Delta(G)+1$*, ahol* $\Delta(G)$ *a gráf csúcsai fokszámának maximumát jelöli.*

Definíció: Egy ***csúcs fokszáma*** *a hozzá illeszkedő élek számával egyenlő.*

Definíció: Egy n csúcsot számláló gráf ***teljes****, ha minden csúcspárt él köt össze.*

Egy gráf kromatikus számának szimulált hűtéssel történő felső becslésének vázlata.

Meg kell adni a következőket:

- megengedett megoldások halmaza.
- célfüggvény.
- szomszédsági függvény.
- kezdeti megengedett megoldás.
- hűtési ütemterv.

A megengedett megoldások halmazát a gráf csúcsainak $\Delta(G)+1$ részhalmazból álló $s = (V_1, V_2, \ldots, V_{\Delta(G)+1})$ felbontásai alkotják, ahol V_j halmazok üresek is lehetnek.

A célfüggvény $h(s) = \sum_j w_j(| V_j | - \lambda | E(V_j) |$, ahol w_j súlyok nem negatív számok és szigorúan monoton csökkenő sorozatot alkotnak. A w_j súlyok sorozatának megválasztása tetszőleges lehet. Például valamilyen csökkenő sorozat. Minden j = 1, 2, …, $\Delta(G)+1$-re az $E(V_j)$ a V_j-beli csúcsokat összekötő éleket tartalmazza.

Ezzel megadtunk egy kombinatorikus optimalizálási feladatot, ahol keresni kell egy olyan megengedett megoldást, amelyen a célfüggvény maximumát veszi fel.

A szomszédsági függvényt transzformációs szabályokkal adjuk meg. Egy adott partícióból úgy kapunk szomszédosat, ha egy tetszőleges csúcsot átteszünk egy másik részhalmazba. A *G* gráf csúcsainak más és más részhalmazba való átmozgatásával egy adott $(V_1V_2,...,V_{\Delta(G)+1})$ felbontásból egy tetszőleges másik felbontásba tudunk eljutni, vagyis a gráf erősen összefüggő.

A szomszédsági gráf csúcsainak száma $(\Delta(G)+1)^n$, amit egy *n* halmazból $\Delta(G)+1$ halmazba történő leképezéssel kapunk.

Egy kezdeti megoldás lehet egy véletlen partícionálás, amely csak a 2. feltételét teljesíti a k-színezésnek, vagy egy heurisztikus algoritmussal talált színezés.

A hűtési ütemterv kezdeti hőmérsékletét, valamint a hőmérséklet csökkentését a 28. oldalon szereplő hűtési ütemtervben leírtaknak megfelelően adhatjuk meg. A folyamat hossza egy nagy konstans, amely a gráf csúcsai számának többszöröse.

A tabu-keresés

(Tabu search)

A lokális keresés alapalgoritmusának kisebb módosításával kapjuk a tabu-keresés algoritmusát. Kiindulunk egy aktuális megengedett megoldásból, és jobb megengedett megoldást keresünk. Mindig a legjobb szomszédba lépünk tovább, még akkor is, ha az rosszabb költségű az aktuális megoldásnál. Használunk egy *tabu listát*, amely tárolja, hogy hol járt eddig a keresés, ezáltal meg tudjuk akadályozni, hogy ismét bejárjuk a már bejárt utakat, így a keresés hatékonyabb lesz.

```
 1: procedure Tabu-keresés(<A, kezdő, C, O>, k)
 2: aktuális ← kezdő
 3: legjobb ← aktuális
 4: következő ← NIL
 5: i ← 1
 6: T ← Ø
 7: while IGAZ do
 8:     if MEGÁLLÁSI_FELTÉTEL then
 9:         break
10:     end if
11:     O' ← { o | o ∈ O ∧ ELŐFELTÉTEL(aktuális,o) ∧ ALKALMAZ(aktuális,o) ∉ T }
12:     operátor ← VÁLASZT({ o | o ∈ O' ∧
13:     ∧ ∀o'( o' ∈ O' → k(ALKALMAZ(aktuális,o)) < k(ALKALMAZ(aktuális,o')))})
14:     következő ← ALKALMAZ(aktuális,operátor)
15:     T ← T ∪ {következő}
16:     aktuális ← következő
17:     if k(következő) < k(legjobb) then
18:         legjobb ← következő
19:     end if
20:     i ← i+1
21: end while
22: print legjobb
23: end procedure
```

Az algoritmus magyarázata

Ahogy az előző algoritmusok esetén, most is egy szomszédsági függvény által kifeszített szomszédsági gráfon dolgozunk, melynek csomópontjai megengedett megoldások. A 2-6. pont tartalmazza az inicializációs részt. A 2. pontban az *aktuális* csomópont felvesz egy *kezdő* állapotot. Használni fogunk egy *legjobb* állapotot, amely kezdetben a *kezdő* állapotot veszi fel értékül (3. pont). A keresés során mindig ez fogja tárolni a legjobb megengedett megoldást. A *következő* csomópont a programban mindig az *aktuális* csomópont legjobb szomszédját fogja felvenni értékül. Használunk egy *k* változót, aminek értéke az iterációk számával fog megegyezni, valamint egy *T* halmazt, amely megengedett megoldásokat tartalmaz. Ezt a *T* halmazt **rövid távú memóriának** nevezzük, és ez vezérli a keresést.

Maga a keresés egy ciklus, melyet a 7. pontban indítunk, és akkor áll meg, ha teljesül a MEGÁLLÁSI_FELTÉTEL (8. pont). A 11-13. pontban meghatározzuk az aktuális csomópont legjobb szomszédját az *SZF(aktuális,x) – T* halmazból. Ez azt jelenti, hogy az *aktuális* állapot *SZF()* szomszédsági függvény által meghatározott szomszédai közül választunk, mely szomszédok nem szerepelhetnek a *T* halmazban, vagyis a **rövid távú memóriában.** Ez a feltétel a 11. pontban található: ALKALMAZ(*aktuális,o*) $\notin$ *T*. A 14. pontban a *következő* csomópont felveszi az *aktuális* csomópont előző pontokban meghatározott legjobb szomszédját. A 15. pontban a **rövid távű memóriába** felvesszük a *következő* csomópontot. Tehát meghatároztuk a következő állapotot, ahol folytatódni fog a keresés. A 17-19. pontban megnézzük, hogy ez az új *aktuális* megengedett megoldás jobb-e, mint az eddigi *legjobb* megengedett megoldás:

- ha igen, akkor a *legjobb* csomópont felveszi az új *aktuális* csomópontot értékül.
- ha nem, akkor a *legjobb* értéke változatlan marad.

A jelenlegi feladat minimalizálási feladat, amit a *k*(*következő*) < *k*(*legjobb*) formulában a < reláicó jelez. Maximalizálási feladatnál a reláicó megfordul. A 20. pontban növeljük az *i* változó értékét 1-el. Ennek a változónak a MEGÁLLÁSI_FELTÉTEL-ben van szerepe. A 22. pontban kiírjuk a legjobb megengedett megoldást.

Megállási (terminálási) feltételek:

- az *aktuális* megengedett megoldás optimális megoldás.
- az *i* változó értéke (iterációk száma) túllép egy bizonyos határt.
- az iterációk száma a *legjobb* csomópont utolsó módosítása óta túllép egy bizonyos határt. Ez úgy oldható meg, hogy bevezetünk egy i^* változót, amely felveszi *i* értékét a *legjobb* csomópont utolsó módosulásakor, majd a megállási feltételben megvizsgáljuk a i-i^* különbséget
- az *SZF(aktuális,x) – T* halmaz üres. Ha ez teljesül, a keresés nem folytatható.

A rövid távú memória szerepe a keresésben

Az algoritmus nem feltétlenül akad meg, ha az *aktuális* megengedett megoldás lokálisan optimális. Ha az *aktuális* megengedett megoldás *SZF(aktuális,x) – T* szomszédsága nem üres, és minden, ebbe a halmazba tartozó *következő* szomszédjára igaz minimalizálási feladat esetén, hogy $k(aktuális) \leq k(következő)$ (ami azt jelenti, hogy az *aktuális* csomópont lokálisan optimális), akkor ez igaz lesz a legjobb *következő* szomszédjára is. Így előfordulhat, hogy keresés során visszalépünk egy már vizsgált megengedett megoldásba, és onnan újra ide, vagyis ciklusba kerülünk. A fenti probléma kezelésére alkalmazzuk a **rövid távú memóriát** (*T halmaz*), amely tartalmazza a legújabb megengedett megoldások egy részét. Ha a memória az utolsó *t* darab megengedett megoldásra emlékezik, akkor az algoritmus elkerüli a legfeljebb *t* hosszúságú ciklusok kialakulását. Hogy szem előtt tartsuk az algoritmus hatékonyságát, a **rövid távú memória** általában a legutóbb használt **operátorok inverzét** tartalmazza. Tehát van egy operátor, melyet alkalmazva az *aktuális* csomópontra, megkapjuk a *következő* csomópontot. Az operátor inverze az az operátor lesz, melyet alkalmazva a *következő* csomópontra, megkapjuk az *aktuális* csomópontot, vagyis visszalépünk az előző állapotba. Ez az inverz operátor fog bekerülni a *T* halmazba, és a *következő* csomópont választásakor ezért szűkítjük a választható csomópontokat *SZF(aktuális,x) – T*-re, hogy ne tudjunk vissza lépni egy korábbi állapotba, vagyis a *T*-beli inverz műveletek **tabu státuszt** kapnak. Innen ered az algoritmus neve. A memória **felejteni** is tud. Előfordulhat, hogy egy operátor több megengedett megoldásra is alkalmazható, és ha elég távolra kerültünk az

aktuálistól, akkor az inverz operátor már hasznos lehet. Használhatunk *t* hosszúságú **tabu listát**, ahol az éppen tiltásra került operátor a lista elejére kerül, majd ha a lista meghaladja a *t* hosszúságot, akkor a legvégéről töröljük a legrégebbi operátort. A tabu keresés intenzitása növelhető, ha keresés során a szomszédsági gráfnak csak egy bizonyos részére koncentrálunk. Például megkeressük a legjobb megengedett megoldások közös tulajdonságait, és nem fogadunk el olyan aktuális megengedett megoldást, mely nem rendelkezik ilyen tulajdonságokkal. Ezen tulajdonságokra való emlékezés **középtávű memória** megvalósítását jelenti. Ha feltártuk az előbbi részgráfot, akkor átugorhatunk a gráf egy távolabbi pontjába, és ott folytathatjuk a keresést. Ebben az esetben **hosszútávú memóriáról** beszélünk.

Gráfszínezés tabu-kereséssel

Cél: egy gráf kromatikus számának meghatározása.

Legyenek adva a *„Gráfszínezés szimulált hűtéssel”* című részben (28. oldal) megadott tételek, definíciók és jelölések.

A megengedett megoldások halmazának, valamint a célfüggvény megadásával definiálunk egy kombinatorikus optimalizálási feladatot:

- Megengedett megoldások halmaza: a *G* gráf csúcsainak *k* részhalmazából álló *aktuális*=(V_1, V_2, …, V_k) partíciók
- Célfüggvény: $h(aktuális) = \sum_j | E(V_j) |$

A *G* gráf pontosan akkor színezhető *k* színnel, ha van olyan *legjobb*=(V_1, V_2, …, V_k) partíció, melyre $h(legjobb)=0$.

A szomszédsági függvényt operátorokkal adjuk meg, ahol az *o* operátor egy kiválasztott *cs* csomópontot kivesz a $cs \in V_i$ halmazból, és átteszi V_j–be, melyre $i \neq j$.

A tabu-keresés során, amikor egy operátor egy *cs* csomópontot kivesz a V_i részhalmazból, akkor a (*cs,i*) pár tabu listára kerül. Tehát az inverz (tiltott) operátor a *cs* csomópontot V_i -be

próbálja tenni. Ekkor viszont elérhetetlenné válhatnak az eddigieknél jobb megoldások is. Ennek kezelésére találták ki az **aspirációs feltételt**, ami a következő alfejezet témája.
A megállási feltétel legyen a *h(aktuális)=0* **or** $k > kmax$. A ciklus akkor áll meg, ha ez teljesül.

Ha az algoritmus lefutása után *h(legjobb)=0*, akkor a gráf kiszínezhető legfeljebb k színnel. Ellenkező esetben a válasz bizonytalan.

```
 1: procedure Gráf_kromatikus_számának_felső_becslése(<A, kezdő, C, O>, h)
 2:     L ← 1
 3:     U ← Δ(G)+1
 4:     while L<U do
 5:         k ← (L+U)/2
 6:         if G_kiszinezhető_k_színnel then
 7:             U ← k
 8:         else
 9:             L ← k+1
10:         end if
11:     end while
12:     print L
13:     end procedure
```

Az algoritmusban az L egy alsó, az U pedig egy felső érték (2-3. pont), ahol U kezdetben a G gráf csúcsai fokszámának a maximuma plusz egy.

A ciklus addig fog futni, amíg L kisebb, mint U (4. pont). L értékét k függvényében fogjuk növelni. Az 5. pontban k úgy vesz fel értéket, hogy arra mindig teljesüljön az $L \leq k < U$.
Ha G gráf kiszínezhető k színnel, akkor az U k-t veszi fel értékül, egyébként növeljük az L értéket. Az L értéke lesz a gráf kromatikus számának felső becslése.
Az algoritmus 6. pontjában a feltétel helyén az előző oldalakon tárgyalt tabu-kereséssel vizsgálhatjuk meg a gráf k színnel való kiszínezhetőségét. Tehát a feltétel egy függvény, mely visszatérési értékkel szolgál, a függvény törzse pedig a tabu-keresés.

Aspirációs feltétel

Mivel egy operátor több csomópontra is alkalmazható, ezért előfordulhat, hogy ha tabu listára kerül, akkor nem fogunk tudni megtalálni egy eddigieknél jobb megengedett megoldást. Az **aspirációs szinttel** elégséges feltételt tudunk adni arra, hogy ha egy operátor tabu listán van, akkor csak abban az esetben alkalmazhassuk egy megengedett megoldásra, ha korábban még biztosan nem alkalmaztuk rá.

Egy *aktuális* megengedett megoldás **aspirációs szintje** az a legkisebb (maximalizálási feladat esetén legnagyobb) *h(ALKALMAZ(cs,operátor'))* érték, amelyet a korábban vizsgált *cs* csomópont vizsgálata során az algoritmus talált és melyre *h(aktuális)=h(cs)*. Az *operátor'* a *cs* csomópontra korábban alkalmazott művelet.

Egy *következő* csomópont teljesíti az **aspirációs feltételt**, ha *h(következő)<A(h(aktuális),* ahol az *A()* az aspirációs szintet jelöli.

Ha az algoritmus a *cs* csomópont valamely *cs'* szomszédját választja következő megengedett megoldásnak, akkor az aspirációs szintet a következő értékadással kell aktualizálni:

A(h(cs)) = min(A(h(cs)), h(cs')).

Vagyis az aktuális megengedett megoldás, és a kiválasztott következő megengedett megoldás aspirációs szintje közül a kisebb érték lesz az új aspirációs szint.

A *h(következő) < A(h(aktuális))* aspirációs feltétel teljesülése garantálja, hogy a *következő* csomópontot korábban még nem vizsgáltuk.

Most nézzük meg az aspirációs feltételt algoritmus segítségével.

Az alábbi algoritmus a *következő* csomópontot határozza meg. Magát az algoritmust a fejezet elején tárgyalt tabu-keresés algoritmusában a 11-14. pontban található utasítások helyére másolhatjuk be. Az ott lévő utasításokat töröljük.

```
1: következő ← NIL
2: O' ← { o | o ∈ O ∧ ELŐFELTÉTEL(aktuális,o) }
3: for all o ∈ O' do
4:     if ALKALMAZ(aktuális,o)∉ T or h(ALKALMAZ(aktuális,o))<A(h(aktuális)) then
5:         if következő=NIL or h(ALKALMAZ(aktuális,o))<h(következő) then
6:             következő ← ALKALMAZ(aktuális,o)
7:         end if
```

```
 8:              if h(ALKALMAZ(aktuális,o))<h(aktuális) then
 9:                      következő ← ALKALMAZ(aktuális,o)
10:                      break
11:              end if
12:      end if
13: end for
14: A(h(aktuális)) = min(A(h(aktuális)), h(következő))
```

A fenti programrész a tabu-keresés ciklusában az *aktuális* megengedett megoldás legjobb szomszédját keresi meg. A 2. pontban felvesszük *O'* halmazba az *aktuális* megengedett megoldásra alkalmazható összes operátort. A 3. pontban indítunk egy ciklust, amely a ciklus magjában lévő utasításokat végre hajtja az *O'*-beli összes operátorra. A 4. pontban a feltétel első része kiköti, hogy az operátor nem lehet inverz operátor, vagyis olyan, melyet alkalmazva az *aktuális* megengedett megoldásra, egy olyan megengedett megoldást kapunk, mely szerepel a *T* halmazban. A feltétel második része azt vizsgálja, hogy a kiválasztott operátort alkalmazva az *aktuális* csomópontra, a kapott csomópont költsége kisebb-e az *aktuális* csomópont aspirációs szintjénél. Az **if**-es feltétel akkor lesz igaz, ha a két részfeltétel közül legalább az egyik igaz. Az 5-7. pont az *aktuális* csomópont szomszédai közül a legjobbat keresi. A jobb szomszéd lesz mindig a *következő* megengedett megoldás. A 8-10. pontban, ha az aktuális operátort alkalmazva az *aktuális* csomópontra olyan szomszédot kapunk, melynek költsége jobb, mint az *aktuális* csomópont költsége, akkor ez a szomszéd lesz az új *aktuális* csomópont, és befejeztetjük a **for all** ciklust. A 14. pontban aktualizáljuk az aspirációs szintet.

Összefoglalás

A dolgozatban bemutattam az IDA* algoritmust, a Tabu-keresést, a Szimulált hűtést, valamint a Lokális keresés alapalgoritmusát.

Az IDA* algoritmus fán vagy gráfon dolgozik, heurisztikát használ, és egy kezdőállapotból célállapotba vezető, minél kisebb költségű utat keres. Az algoritmus hatékonysága nagyban függ a heurisztikus függvény minőségétől. Az IDA* teljes és optimális, feltéve, hogy garantálni tudjuk, hogy *h(n)* elfogadható (fa-keresés esetén) vagy monoton (gráf-keresés esetén). Kevesebb memóriát használ az A* algoritmusnál.

A lokális keresési algoritmusok nem keresnek és nem is tárolnak utat. Heurisztikát sem használnak. Csupán egyetlen állapotot tárolnak a memóriában, ami az aktuális megengedett megoldás. Ebből a megengedett megoldásból próbálnak szomszédsági függvény általi transzformációval egy jobb költségű szomszédba lépni, ami azt jelenti, hogy valamilyen módosítást hajtanak végre az aktuális megengedett megoldáson. A lokális keresők hatékonysága nagyban függ a megválasztott szomszédsági függvénytől. Nem garantálják az optimális megoldás megtalálását, csak közelítik azt. A lokális keresők beragadhatnak egy lokális minimumban. A szimulált hűtés megfelelő hűtési ütemterv esetén el tudja kerülni a lokális minimumokat, és optimális megoldást talál, feltéve, hogy megfelelő a szomszédsági függvény. A Tabu keresés segítségével elkerülhetjük a hurkokat, amik az alapalgoritmus esetén fordulhatnak elő. Ezt úgy oldja meg, hogy használ egy rövid távú memóriát, amibe elhelyezi a már bejárt csomópontokat. Az algoritmus futás során nem léphet vissza olyan csomópontba, ami szerepel ebben a rövid távú memóriában

A függelékben található egy „legrövidebb út” feladat, valamint annak Java nyelvű implementálása, és az algoritmusok futási eredménye a feladatra. Ezeket a futási eredményeket gyűjtöttem most ide egy táblázatba.

	IDA*	Lokális keresés alapalgoritmusa	Tabu-keresés	Szimulált hűtés
Költség	92	165	104	111
Időigény	23	10	20	6
Tárigény	9	1	21	1
Optimalitás	igen	nem	nem	nem
Csomópontok száma a megoldásban	5	7	5	5
Élek száma a megoldásban	4	6	4	4

Az IDA* algoritmus optimális, tehát egyedül ő találta meg a legrövidebb utat. A lokális keresők nem találtak optimális megoldást. Az alapalgoritmus költsége feltünően magas, tehát nem túl hatékony erre a feladatra. Ennél jobb megoldást talált mindkét javítása, a Tabu-keresés és a Szimulált hűtés. Ez azért történt, mert amíg az alapalgoritmus csak jobb megengedett megoldásokba léphet át, addig a másik kettő rosszabba is. Mivel a kezdeti megoldás első három csomópontja *Nyíregyháza−>Nagyhalász−>Kisvárda* út, és vezet él Nyíregyháza és Kisvárda között, ezért a szomszédsági függvény összekötheti őket, ami Nagyhalász kizárását jelentené a körből. Ekkor viszont a kör költsége nagyobb lenne, ezért az alapalgoritmus ezt a lépést nem engedi meg, a Szimulált hűtés és a Tabu-keresés viszont igen. Ezért találnak jobb megoldást, mert az új (drágább) körön később létre tudnak hozni olyan módosítást, amit az eredeti körön az alapalgoritmus nem tud.

Időigény szempontjából a Szimulált hűtés jobb a másik két lokális keresőnél. Az időigény alapegysége egyetlen transzformációs lépés az aktuális megoldáson. Ebből a szempontból mindhárom lokális kereső hatékonyabb az IDA*-nál. Utóbbinál az időigény a kiterjesztett csomópontok számával egyezik meg.

A lokális keresők tárigénye alacsonyabb, mint az IDA*-é. Ez a másik előnyük vele szemben. A tárigény a memóriában tárolt csomópontok maximális számát mutatja. Lokális keresőknél azért egy, mert csak az aktuális állapotot tárolják. Viszont a Tabu-keresésnél hozzá adódik a rövidtávú memória tartalma is, ami jelen esetben húsz csomópontot tartalmaz. Célszerű lehet korlátozni a rövidtávú memória kapacitását, hogy tárigény szempontjából hatékonyabbá tegyük az algoritmust. IDA*-nál nem egy állapotot tárolunk, hanem azon csomópontokat, amelyek a kezdőállapotból a célállapotba vezető út mentén találhatók.

A táblázat utolsó két sora azt mutatja, hogy a megtalált útvonalon hány város és útszakasz található. Ebből a szempontból a lokális keresés alapalgoritmusa adja a legrosszabb eredményt, míg a másik három kereső eredménye megegyezik.

Ebben a feladatban a legkisebb költségű algoritmus versenyt az IDA* nyerte, a legrövidebb futási idejű, valamint a legkevesebb memóriát igénylő algoritmus pedig a Szimulált hűtés lett.

Irodalomjegyzék

Könyvek:

- Futó Iván: Mesterséges Intelligencia, 2003, Aula Kiadó
- Stuart Russel, Peter Norvig: Mesterséges Intelligencia Modern megközelítésben, 2005, Panem Könyvkiadó
- Stuart Russel, Peter Norvig: Mesterséges Intelligencia Modern megközelítésben, 2000, Panem Könyvkiadó
- Starkné Werner Ágnes: Mesterséges Intelligencia, 2004, Veszprémi Egyetemi Kiadó

Jegyzetek:

- Dombi József: Mesterséges Intelligencia 2, 2007, Szegedi Tudományegyetem

Internetes források:

- Dr. Várterész Magda: Mesterséges Intelligencia 1 előadások, 2006/2007, Debreceni Egyetem
  http://www.inf.unideb.hu/~varteres/mi1folia/foliafo.pdf
  (2008.04.17.)
- Dr. Szalay Tibor: Informált keresés
  http://www.manuf.bme.hu/gdf/2007_Legjobbat_eloszor.pdf
- Mikó Balázs: Mesterséges Intelligencia, 2000
  http://free.x3.hu/vinyisoft/science/mesterseges%20int.pdf
  (2008.04.17.)
- Dr. Kovács Szilveszter: Intelligens Rendszerek I., Problémamegoldás kereséssel
  http://users.iit.uni-miskolc.hu/~szkovacs/GAMFIR/IRE6.pdf
  (2008.04.17.)
- Maliosz Markosz: Hálózattervezési módszerek, eszközök, algoritmusok 2., 2008
  http://opti.tmit.bme.hu/ihsz/2008/MM/ikhsz_ea12_p4.pdf
  (2008.04.17.)

- Csanád Imre: Lokális kereső algoritmusok ütemezési problémákra
  http://www.inf.u-szeged.hu/~cimreh/lok.pdf
  (2008.04.17.)
- Herczeg Artúr: A Tabu keresés
  http://techies.teamlupus.hu/wp-content/uploads/Documents/Tabukereses.pdf
  (2008.04.17.)
- Keld Helsgaun: An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic
  http://www.akira.ruc.dk/~keld/Research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf
  (2008.04.17.)
- Dr. Gregorics Tibor: Vezérlési stratégiák, ELTE
  http://people.inf.elte.hu/gt/mi/kereses/kereses.pdf
  (2008.04.17.)

Függelék

Legrövidebb út probléma

Adott egy úthálózat, amit a következő oldalon találunk.

Nyíregyházától indulva Fehérgyarmatba akarunk eljutni a legrövidebb úton. Az algoritmusok ezt a legrövidebb utat fogják megkeresni, ha tudják. Az úthálózaton a városok csomóponttal vannak jelölve és élekkel (út) vannak összekötve. Az éleken szereplő számok a két város közötti távolságokat mutatják.

Az IDA* heurisztikát is használni fog. Mivel a célváros Fehérgyarmat, ezért a heurisztika Fehérgyarmat és a városok közötti távolság légvonalban, amit a következő táblázatban adok meg.

Távolság légvonalban Fehérgyarmattól

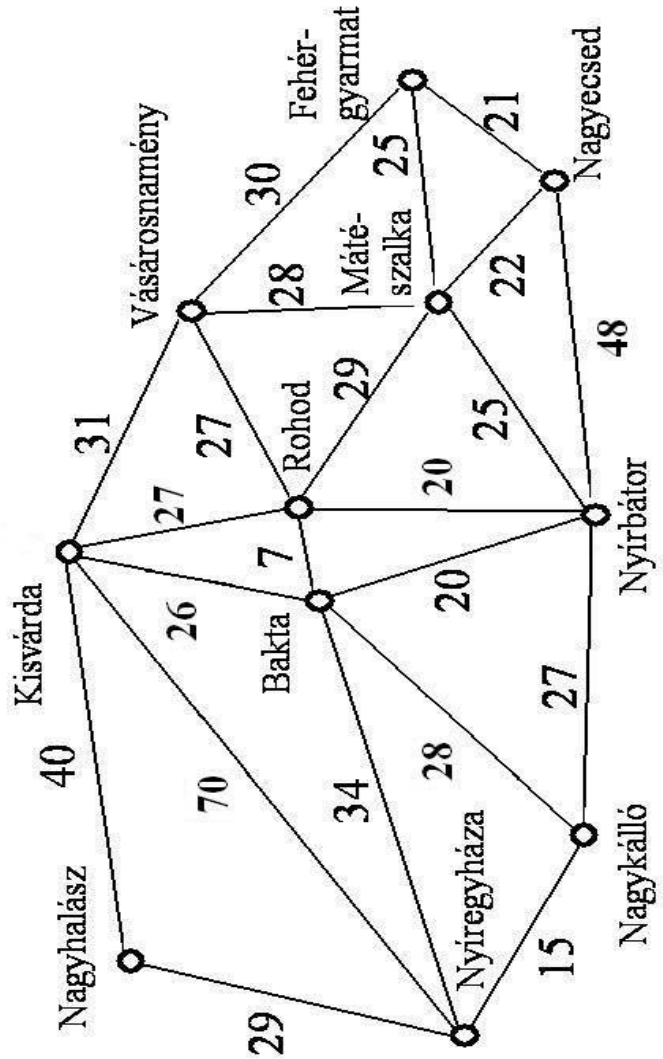
Nyíregyháza	58 kilométer
Nagyhalász	60 kilométer
Kisvárda	40 kilométer
Nagykálló	53 kilométer
Vásárosnamény	22 kilométer
Rohod	27 kilométer
Bakta	34 kilométer
Nyírbátor	32 kilométer
Mátészalka	13 kilométer
Nagyecsed	16 kilométer

Az IDA* algoritmus esetén a költség = 'megtett út hossza' + heurisztika.
Például, ha Rohodon vagyunk, ahová Baktán keresztül érkeztünk Nyíregyházáról, akkor:
költség =(Nyíregyháza−>Bakta + Bakta −>Rohod)+(Rohodtól légvonal Fehérgyarmatig)
költség = (34+7) + 27 = 68.

A lokális keresőknél a költség = 'megtett út hossza'. A három lokális keresőhöz megadtam egy kezdeti megengedett megoldást, egy kört Nyíregyházától Fehérgyarmatig: *Nyíregyháza−>Nagyhalász−>Kisvárda−>Vásárosnamény−>Rohod−>Bakta−> Nagykálló−>Nyírbátor−>Mátészalka−>Nagyecsed−>Fehérgyarmat−>Nyíregyháza*.

Az algoritmusoknál használt szomszédsági függvény ezt a kört fogja transzformálni úgy, hogy a kör i+1-edik csomópontját elhagyja, amennyiben éllel összeköthető az i-edik és i+2-edik csomópont. Például a feladat kezdeti megengedett megoldásának első három eleme *Nyíregyháza−>Nagyhalász−>Kisvárda.*

A gráfon jól látható, hogy Nyíregyházát és Kisvárdát éllel köthetem össze, tehát egy transzformáció hatására az előbbi út *Nyíregyháza−>Kisvárdára* változik, vagyis *Nagyhalász* kikerül a körből. A transzformációk hatására a kör egyre kisebb, az út pedig egyre rövidebb lesz.



Az IDA* algoritmus forráskódja

```
public abstract class Operator{}

public class Utaz extends Operator{

	private String honnan,hova;
	private int tavolsag,heurisztika;

	public Utaz(String a,String b,int c,int d){
		honnan=a;
		hova=b;
		tavolsag=c;
		heurisztika=d;
		}

	public String toString(){
		return("["+honnan+","+hova+"]");
	}

	public String getHonnan(){ return honnan;}
	public String getHova(){ return hova;}
	public int getTavolsag(){return tavolsag;}
	public int getHeurisztika(){return heurisztika;}
}

import java.util.*;

public class Csomopont{
	private Ut ut;
	private Csomopont szulo;
	private int koltseg,uthossz;
	private Operator operator;
	private Set<Operator> alkOperatorok;

	public Csomopont getSzulo(){ return szulo; }
	public int getKoltseg(){ return koltseg;}
	public Operator getOperator(){ return operator;}
	public Set<Operator> getAlkOperatorok(){ return alkOperatorok;}
	public Ut getUt(){ return ut;}
	public int getUthossz(){return uthossz;}

	// A start csucsot ez a konstruktor fogja inicializalni
	public Csomopont(Ut s){
		ut=s;
		szulo=null;
		operator=null;
		uthossz=0;
		koltseg=58;
		alkOperatorok=new HashSet<Operator>();

		for(Iterator<Operator> it=Ut.getOperatorok().iterator();
		it.hasNext();)

			Operator o=it.next();
			if(ut.eloFeltetel(o))
				alkOperatorok.add(o);
		}
	}

	// Ez a konstruktor az osszes tobbi csucs tulajdonsagait allitja be
	public Csomopont(Csomopont cs,Operator op){
		szulo=cs;
```

```
            ut=new Ut(cs.getUt(),op);
            operator=op;
            Utaz f=(Utaz) op;
            uthossz=f.getTavolsag()+cs.getUthossz();
            koltseg=uthossz+f.getHeurisztika();
            alkOperatorok=new HashSet<Operator>();

            for(Iterator<Operator> it=Ut.getOperatorok().iterator();
            it.hasNext();)
            {
                  Operator o=it.next();
                  if(ut.eloFeltetel(o))
                        alkOperatorok.add(o);
            }
      }

      public String toString(){
            return operator+", "+ut;}

      public boolean equals(Object obj){
            if(obj==null || !(obj instanceof Csomopont)) return false;
            Csomopont cs=(Csomopont) obj;
            return (ut.equals(cs.getUt()));}
}



import java.util.*;

public class IDA{

      private LinkedList<Csomopont> nyilt=new LinkedList<Csomopont>();
      public LinkedList<Csomopont> getNyilt(){return nyilt;}

      Csomopont temp,kivalasztott;
      int kh,idoigeny=0,tarigeny=0;
      static int csszam=0;
      int kh_uj=30000;

      public IDA(Ut h){
            nyilt.addLast(new Csomopont(h));
            temp=nyilt.getLast();
            kh=nyilt.getLast().getKoltseg();
            }

      public void keres(){
            while (true){

                  if (nyilt.isEmpty()) break;
                  kivalasztott=nyilt.removeLast();
                  if (kivalasztott.getUt().celAllapot())
                        break;
                  if(kivalasztott.getKoltseg()<=kh){
                        kiterjeszt(kivalasztott);
                        idoigeny++;}
                  else
                        if(kivalasztott.getKoltseg()<kh_uj)
                              kh_uj=kivalasztott.getKoltseg();

                  if(nyilt.isEmpty()){
                        if(kh_uj==30000)
                              break;
                        kh=kh_uj;
                        kh_uj=30000;
                        nyilt.add(temp);
```

```
                        }
                }}

        public void kiterjeszt(Csomopont cs){
                for (Operator op:cs.getAlkOperatorok()){
                        Csomopont uj=new Csomopont(cs,op);
                                nyilt.addLast(uj);
                }
                if(nyilt.size()>tarigeny) tarigeny=nyilt.size();
        }


    public static void megoldas(Csomopont cs){
        if(cs!=null){
                csszam++;
                megoldas(cs.getSzulo());
                if (cs.getSzulo()!=null)
                        System.out.print(" - "+cs.getUt());
                else
                        System.out.print(cs.getUt());

                if(cs.getUt().celAllapot())
                        System.out.println("\n\nMegtett ut hossza (koltseg):
                        "+cs.getUthossz());
        }}

    public static void main(String[] args){

        IDA sz=new IDA(new Ut());
        sz.keres();

        if(sz.kivalasztott.getUt().celAllapot()){
                System.out.print("\nA megoldas: ");
                megoldas(sz.kivalasztott);}

        System.out.println("\nA kiterjesztett csomopontok szama (idoigeny):
        "+sz.idoigeny);
        System.out.println("\nTarolt csomopontok maximalis szama (tarigeny):
        "+sz.tarigeny);
        System.out.println("\nCsomopontok szama a megoldasban: "+sz.csszam);
        System.out.println("\nElek szama a megoldasban: "+
        (sz.csszam-1));
}}
```

Lokális keresőalgoritmusok

```
// Ezt az osztalyt mindharom lokalis kereso hasznalja
// Az uthalozat adatait tartalmazza
public class Inicializal{

            private int[] utvonal=new int[11];
            private int[][] kts=new int[22][3];

            public Inicializal(){

                  utvonal[0]=0; utvonal[1]=4; utvonal[2]=3; utvonal[3]=7;
                  utvonal[4]=6; utvonal[5]=2; utvonal[6]=1; utvonal[7]=5;
                  utvonal[8]=9; utvonal[9]=8; utvonal[10]=10;

                  kts[0][0]=0; kts[0][1]=4; kts[0][2]=29;
                  kts[1][0]=0; kts[1][1]=2; kts[1][2]=34;
                  kts[2][0]=0; kts[2][1]=3; kts[2][2]=70;
                  kts[3][0]=0; kts[3][1]=1; kts[3][2]=15;
                  kts[4][0]=1; kts[4][1]=2; kts[4][2]=28;
                  kts[5][0]=1; kts[5][1]=5; kts[5][2]=27;
                  kts[6][0]=2; kts[6][1]=3; kts[6][2]=26;
                  kts[7][0]=2; kts[7][1]=6; kts[7][2]=7;
                  kts[8][0]=2; kts[8][1]=5; kts[8][2]=20;
                  kts[9][0]=3; kts[9][1]=4; kts[9][2]=40;
                  kts[10][0]=3; kts[10][1]=6; kts[10][2]=27;
                  kts[11][0]=3; kts[11][1]=7; kts[11][2]=31;
                  kts[12][0]=5; kts[12][1]=6; kts[12][2]=20;
                  kts[13][0]=5; kts[13][1]=9; kts[13][2]=25;
                  kts[14][0]=5; kts[14][1]=8; kts[14][2]=48;
                  kts[15][0]=6; kts[15][1]=7; kts[15][2]=27;
                  kts[16][0]=6; kts[16][1]=9; kts[16][2]=29;
                  kts[17][0]=7; kts[17][1]=9; kts[17][2]=28;
                  kts[18][0]=7; kts[18][1]=10; kts[18][2]=30;
                  kts[19][0]=8; kts[19][1]=9; kts[19][2]=22;
                  kts[20][0]=8; kts[20][1]=10; kts[20][2]=21;
                  kts[21][0]=9; kts[21][1]=10; kts[21][2]=25;}

            public int[] getUtvonal(){ return utvonal;}
            public int[][] getKts(){ return kts; }
}
```

A lokális keresés alapalgoritmusa

```
import java.util.*;

public class LokalisAlap{

      private int i,idoigeny=0;
      private String[] varosok={"Nyiregyhaza","Nagykallo",
      "Bakta","Kisvarda","Nagyhalasz","Nyirbator",
      "Rohod","Vasarosnameny","Nagyecsed","Mateszalka","Fehergyarmat"};
      private int[] uj_utvonal=new int[11];
      private int[] utvonal=new int[11];
      private Inicializal init;
```

```
// Ez a konstruktor beallitja a kezdoallapotot
public LokalisAlap(){
       init=new Inicializal();
       System.arraycopy(init.getUtvonal(),0,utvonal,0,11);
       }

public void keres(){
       while(true){

              SZF();
              if(koltseg(uj_utvonal)<koltseg(utvonal))
                     System.arraycopy(uj_utvonal,0,utvonal,0,11);
              else
                     break;
       }
}

// A kovetkezo fuggveny a parameterben megkapott ut
// koltseget szamolja ki
public int koltseg(int[] ut){
       int szamol=0;
       for(int z=0;z<ut.length-1;z++)
       for(int j=0;j<22;j++){
              if((init.getKts()[j][0]==ut[z] && init.getKts()[j][1]==ut[z+1])
              ||
       (init.getKts()[j][0]==ut[z+1] && init.getKts()[j][1]==ut[z]))
                     szamol=szamol+init.getKts()[j][2];}
       return szamol;
}

// A szomszedsagi fuggveny kovetkezik
void SZF(){
       int temp=koltseg(utvonal);
       for(i=0;i<9;i++)
              for(int j=0;j<22;j++)
                     if((init.getKts()[j][0]==utvonal[i] &&
                     init.getKts()[j][1]==utvonal[i+2]) ||
                     (init.getKts()[j][1]==utvonal[i] &&
                     init.getKts()[j][0]==utvonal[i+2])){
                            idoigeny++;
                            uj_utvonal=new int[11];
                            System.arraycopy(utvonal,0,uj_utvonal,0,i+1);
System.arraycopy(utvonal,i+2,uj_utvonal,i+1,utvonal.length-(i+2));
                            if(koltseg(uj_utvonal)<koltseg(utvonal))
                            return;
                                   else
                            break;
                            }

}

// Eredmenyek kiirasaert felelos eljaras kovetkezik
public void kiir(){
       System.out.print("\nA megoldas: "+varosok[0]);
       for(i=1;utvonal[i]!=0;i++)
              System.out.print(" - "+varosok[utvonal[i]]);
       System.out.println();
       System.out.println("\nMegtett ut hossza (koltseg): "+
       koltseg(utvonal));
       System.out.println("\nTranszformaciok szama (idoigeny): "+
       idoigeny);
       System.out.println("\nTarolt csomopontok maximalis szama (tarigeny):
       "+1);
       System.out.println("\nCsomopontok szama a megoldasban: "+i);
       System.out.println("\nElek szama a megoldasban: "+(i-1));
```

```
	}

	public static void main(String[] args){

		LokalisAlap sz=new LokalisAlap();
		sz.keres();
		sz.kiir();
}}
```

A tabu keresés forráskódja

```
import java.util.*;

public class Tabu{

	private int i,k,idoigeny=0;
	private String[] varosok={"Nyiregyhaza","Nagykallo","Bakta",
	"Kisvarda","Nagyhalasz","Nyirbator","Rohod",
	"Vasarosnameny","Nagyecsed","Mateszalka","Fehergyarmat"};
	private int[] utvonal=new int[11];
	private int[] uj_utvonal=new int[11];
	private int[] legjobb_utvonal=new int[11];
	private Vector<int[]> T=new Vector<int[]>();
	private Inicializal init;

	// Kezdoallapot beallitasa konstruktorral
	public Tabu(){
		init=new Inicializal();
		System.arraycopy(init.getUtvonal(),0,utvonal,0,11);
		System.arraycopy(utvonal,0,legjobb_utvonal,0,11);
		k=1;
		}

	public void keres(){
		while(true){
			if(k>20) break;
			SZF();
			T.add(uj_utvonal);
			System.arraycopy(uj_utvonal,0,utvonal,0,uj_utvonal.length);

			if(koltseg(uj_utvonal)<koltseg(legjobb_utvonal))
	System.arraycopy(uj_utvonal,0,legjobb_utvonal,0,uj_utvonal.length);
			k++;
	}}

	// A kovetkezo fuggveny a parameterben megkapott ut
	// koltseget szamolja ki
	public int koltseg(int[] ut){
		int szamol=0;
		for(int z=0;z<ut.length-1;z++)
		for(int j=0;j<22;j++){
			if((init.getKts()[j][0]==ut[z] && init.getKts()[j][1]==ut[z+1])
			||
			(init.getKts()[j][0]==ut[z+1] &&
			init.getKts()[j][1]==ut[z]))
				szamol=szamol+init.getKts()[j][2];}
		return szamol;
	}
```

```
	// A szomszedsagi fuggveny kovetkezik
	void SZF(){
		int[] temp_ut=new int[11];
		System.arraycopy(utvonal,0,temp_ut,0,utvonal.length);
		int legjobb_koltseg=1000;
		for(i=0;i<9;i++)
			for(int j=0;j<22;j++)
				if((init.getKts()[j][0]==utvonal[i] &&
				init.getKts()[j][1]==utvonal[i+2]) ||
				(init.getKts()[j][1]==utvonal[i] &&
				init.getKts()[j][0]==utvonal[i+2])){
					idoigeny++;
					uj_utvonal=new int[11];
					System.arraycopy(utvonal,0,uj_utvonal,0,i+1);
		System.arraycopy(utvonal,i+2,uj_utvonal,i+1,utvonal.length-(i+2));
					if(T.contains(uj_utvonal)==false &&
					koltseg(uj_utvonal)<legjobb_koltseg){
							legjobb_koltseg=koltseg(uj_utvonal);
		System.arraycopy(uj_utvonal,0,temp_ut,0,uj_utvonal.length);
					}}
		System.arraycopy(temp_ut,0,uj_utvonal,0,temp_ut.length);
	}


	// Eredmenyek kiirasaert felelos eljaras kovetkezik
	public void kiir(){
		System.out.print("\nA megoldas: "+varosok[0]);
		for(i=1;utvonal[i]!=0;i++)
			System.out.print(" - "+varosok[utvonal[i]]);
		System.out.println();
		System.out.println("\nMegtett ut hossza (koltseg): "+
		koltseg(utvonal));
		System.out.println("\nTranszformaciok szama (idoigeny): "+
		idoigeny);
		System.out.println("\nTarolt csomopontok maximalis szama (tarigeny):
		"+(1+T.size()));
		System.out.println("\nCsomopontok szama a megoldasban: "+i);
		System.out.println("\nElek szama a megoldasban: "+(i-1));
	}


	public static void main(String[] args){

		Tabu sz=new Tabu();
		sz.keres();
		sz.kiir();
}}
```

A szimulált hűtés forráskódja

```
import java.util.*;

public class Szimulalt{

	private int i,k,idoigeny=0;
	private String[] varosok={"Nyiregyhaza","Nagykallo","Bakta",
	"Kisvarda","Nagyhalasz","Nyirbator","Rohod",
	"Vasarosnameny","Nagyecsed","Mateszalka","Fehergyarmat"};
```

```
private int[] utvonal=new int[11];
private int[] uj_utvonal=new int[11];
private int T,L;
private Inicializal init;
// Kezdoallapot beallitasa kovetkezik
public Szimulalt(){
       init=new Inicializal();
       System.arraycopy(init.getUtvonal(),0,utvonal,0,11);
       T=15;
       L=2;
       k=0;
       }

public void keres(){
       while(true){

       if(T<1) break;
       for(int n=1;n<=L;n++){
              SZF();
              if(koltseg(uj_utvonal)<=koltseg(utvonal))
                     System.arraycopy(uj_utvonal,0,utvonal,0,11);
              else
                     if(Math.exp((koltseg(utvonal)-
                     koltseg(uj_utvonal))/T)>Math.random())
                            System.arraycopy(uj_utvonal,0,utvonal,0,11);
       }
       k++;
       T=T-1;
       L=L+2;
}}

// A kovetkezo fuggveny a parameterben megkapott ut
// koltseget szamolja ki
public int koltseg(int[] ut){
       int szamol=0;
       for(int z=0;z<ut.length-1;z++)
       for(int j=0;j<22;j++){
              if((init.getKts()[j][0]==ut[z] && init.getKts()[j][1]==ut[z+1])
              ||
              (init.getKts()[j][0]==ut[z+1] &&
              init.getKts()[j][1]==ut[z]))
                     szamol=szamol+init.getKts()[j][2];}
       return szamol;
}

// Szomszedsagi fuggveny kovetkezik
void SZF(){
       for(i=0;i<9;i++)
              for(int j=0;j<22;j++)
                     if((init.getKts()[j][0]==utvonal[i] &&
                     init.getKts()[j][1]==utvonal[i+2]) ||
                     (init.getKts()[j][1]==utvonal[i] &&
                     init.getKts()[j][0]==utvonal[i+2])){
                            idoigeny++;
                            uj_utvonal=new int[11];
                            System.arraycopy(utvonal,0,uj_utvonal,0,i+1);
System.arraycopy(utvonal,i+2,uj_utvonal,i+1,utvonal.length-(i+2));
                            return;}
}


// Eredmenyek kiirasaert felelos eljaras kovetkezik
public void kiir(){
       System.out.print("\nA megoldas: "+varosok[0]);
       for(i=1;utvonal[i]!=0;i++)
```

```
            System.out.print(" - "+varosok[utvonal[i]]);
            System.out.println();
            System.out.println("\nMegtett ut hossza (koltseg): "+
            koltseg(utvonal));
            System.out.println("\nTranszformaciok szama (idoigeny): "+
            idoigeny);
            System.out.println("\nTarolt csomopontok maximalis szama (tarigeny):
            "+1);
            System.out.println("\nCsomopontok szama a megoldasban: "+i);
            System.out.println("\nElek szama a megoldasban: "+(i-1));
      }

      public static void main(String[] args){

            Szimulalt sz=new Szimulalt();
            sz.keres();
            sz.kiir();
}}
```

Futási eredmények

IDA* algoritmus

```
C:\mestint>java IDA
```

A megoldas: Nyiregyhaza - Nagykallo - Nyirbator - Mateszalka - Fehergyarmat
Megtett ut hossza (koltseg): 92
A kiterjesztett csomopontok szama (idoigeny): 23
Tarolt csomopontok maximalis szama (tarigeny): 9
Csomopontok szama a megoldasban: 5
Elek szama a megoldasban: 4

Lokális keresés alapalgoritmusa

```
C:\mestint>java LokalisAlap
```

A megoldas: Nyiregyhaza - Nagyhalasz - Kisvarda - Bakta - Nyirbator - Mateszalka - Fehergyarmat
Megtett ut hossza (koltseg): 165
Transzformaciok szama (idoigeny):10
Tarolt csomopontok maximalis szama (tarigeny): 1
Csomopontok szama a megoldasban: 7
Elek szama a megoldasban: 6

Szimulált hűtés

```
C:\mestint>java Szimulalt
```

A megoldas: Nyiregyhaza - Nagykallo - Nyirbator - Nagyecsed - Fehergyarmat
Megtett ut hossza (koltseg): 111
Transzformaciok szama (idoigeny): 6
Tarolt csomopontok maximalis szama (tarigeny): 1
Csomopontok szama a megoldasban: 5
Elek szama a megoldasban: 4

Tabu keresés

```
C:\mestint>java Tabu
```

A megoldas: Nyiregyhaza - Bakta - Nyirbator - Mateszalka - Fehergyarmat
Megtett ut hossza (koltseg): 104
Transzformaciok szama (idoigeny): 20
Tarolt csomopontok maximalis szama (tarigeny): 21
Csomopontok szama a megoldasban: 5
Elek szama a megoldasban: 4