**Nem módosítható gráfkereső algoritmusok**

Gráfkeresési probléma

Gráfkeresési probléma alatt azt értjük, hogy van-e olyan út, amely a ’start’ csúcsból valamely terminális csúcsba vezet. Terminális csúcsból több is lehet.

Gráf kereső algoritmusok 2 nagy osztálya:

* módosítható
* nem módosítható

A nem módosítható algoritmusoknál az adatbázist csak az operátorok segítségével változtathatom meg, a módosítható algoritmusoknál az operátoron kívül mindig van legalább egy úgynevezett technikai művelet.

Gráfkereső algoritmusok közös sablonja:

1. adatbázis
   1. ez egy régi szóhasználat, olyan értelemben, hogy mi van a memóriában
   2. korszerűbb szóhasználat lenne az adatszerkezet, mert itt lényegében az adat-szerkezetet kell meghatározni
   3. csak hagyománytiszteletből használjuk az adatbázis szót
2. inicializálás
   1. az adatszerkezetet alkotó változók kezdőértékét itt adjuk
   2. megmondjuk, hogy kezdetben mi van a memóriában
3. ciklus
   1. ez egy új szóhasználat, a régebben a vezérlő szót használtuk
   2. a jegyzetben a vezérlő szót használjuk
   3. ez egy végtelen ciklus, amelyben feltárjuk a gráfot
   4. legeneráljuk a gráfot
   5. mivel a gráf nagy, ezért általában kezdetben nem adott a gráf, hanem operáto-rok segítségével futás közben generálom a gráfot a cikluson belül
   6. ha mégis adott a gráf, akkor általában más jellegű gráfkereső algoritmusokat érdemes használni
4. pozitív kilépési feltétel
   1. mivel a ciklus az egy végtelen ciklus, ezért kilépési feltételekre van szükség
   2. a pozitív kilépési feltétel->megtaláltam a megoldást
   3. azt vizsgálja, hogy megtaláltuk-e a megoldást
   4. praktikusan azt nézzük meg, hogy az aktuális csúcs terminális csúcs-e
5. negatív kilépési feltétel
   1. azt vizsgálja, hogy felismertük-e, hogy nincs megoldás
   2. fel is lehet adni a megoldáskeresést, ilyen esetben nem győződök meg arról, hogy nincs megoldás, csak feladom
6. technikai művelet
   1. pl.: backtrack esetén a visszalépés
   2. mélységi keresés esetén a kiterjesztés
   3. olyan művelet, ami megváltoztatja az adatbázist, de nem operátor

És minden algoritmushoz felteszünk 3 kérdést

1. Ha van megoldás, megtalálja?
2. Ha nincs megoldás, felismeri?
3. Ha talál megoldást, az optimális?
4. Csak akkor tesszük fel, ha a 3. kérdésre igaz a válasz.
   1. Ha optimális, milyen értelemben optimális?

Minden gráfkereső algoritmus erre épül és ezeket a kérdéseket tesszük fel.

Def.: (Megoldás): A gráfkeresési probléma megoldása, 1 út amely a start csúcsból valamely terminális csúcsba vezet.

**Adatbázis**: elavult szóhasználat, jobb lenne az adatszerkezet. Mi van a memóriában, merevlemezen? Milyen változóim vannak?

**Inicializálás**: Változóimnak mi a kezdőértéke?

**Ciklus**: A jegyzetben vezérlő van. A ciklus egy végtelen ciklus, while(true). A gráfkereső algoritmus, amit keresünk, nem feltételezik, hogy kész van a gráf. Úgy írjuk meg, hogy menet közben generálja a gráfot, a ciklusban. A gráf túl nagy, csak az érdekes részét tárom fel. Lusta megközelítés.

A ciklusban feltárom a gráfot. Mivel végtelen ciklus ezért kell a pozitív és a negatív kilépési feltétel.

**Pozitív kilépési feltétel:** Akkor teljesül, ha megtaláltam a megoldást, azaz terminális csúcsot találtam. A terminális csúcs nem a megoldás, csak a terminális csúcsba vezető út.

**Negatív kilépési feltétel:** Amikor azt mondom, hogy nem találtam meg a megoldást. Ez nem jelenti, hogy nincs megoldás, csak én nem találtam

**Technikai művelet:** Nem mindegyiknek van, a nem módosíthatóknál nincs. Arra szolgál, hogy kitudjak jönni a zsákutcából.

**Gráfkereső algoritmusok**

Nem módosítható gráfkereső algoritmusok

* próba-hiba módszer
* hegymászó algoritmus
* mindkettőnek van restartos (újraindítás feladásig) változata: restartos próba-hiba mód-szer, restartos hegymászó algoritmus

Módosítható gráfkereső algoritmusok

* visszalépéses keresés (backtrack)
  + alap backtrack
  + mélység korlátos backtrack
  + körfigyelés backtrack
  + ág-korlátos backtrack
* kereső fával keresők
  + mélységi keresés
  + szélességi keresés
  + best-first keresés (best-first search)
  + optimális keresés
  + A-algoritmus
  + A\*-algoritmus

Műveletvégzés hatása nem vonható vissza. Ez azzal jár, hogy a keresés során „zsákutcába” juthatunk, melyből nem tudunk a keresés egy korábbi állapotába visszajutni. Ezen keresők előnye az egyszerű, kisméretű adatbázis.

A nem módosítható megoldáskeresők jelentősége kisebb, tulajdonságaik miatt ritkán, csak bizonyos problémák esetén használják őket. Az előnyük mindenképpen az, hogy egyszerűek. Csak olyan problémák megoldására használják őket, ahol nem is a megoldás (mint operátorsorozat) előállítása a lényeg, hanem annak eldöntése, hogy létezik-e megoldása a feladatnak, és ha igen, akkor egy (valamilyen) célállapot előállítása.

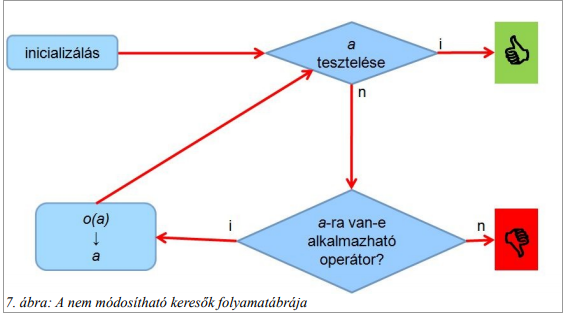
Általános felépítés:

**Adatbázis**: egyetlen állapotból (csúcsból) áll (aktuális állapot).

**Műveletek**: az állapottér-reprezentációban megadott operátorok.

**Vezérlő**: A kezdőállapotra megpróbál egy operátort alkalmazni, és az eredményül kapott állapottal felülírja a kezdőállapotot az adatbázisban. Az új állapotra is próbál operátort alkalmazni, majd ezt az állapotot is felülírja. Ez a ciklikus végrehajtás addig történik, míg az aktuális állapotról ki nem derül, hogy célállapot. Részletesen:

1. *Inicializálás*: A kezdőállapotot elhelyezi az adatbázisban.
2. *Ciklus*:
   1. *Tesztelés*: Ha az aktuális állapot (jelöljük a-val) célállapot, akkor leáll a keresés. Van megoldás.
   2. Van-e olyan operátor, mely alkalmazható a-ra?
      * Ha nincs, akkor leáll a keresés. Nem találtunk megoldást.
      * Ha van, akkor jelöljük el o-val. Legyen o(a) az aktuális állapot.



Tulajdonságai:

**Teljesség**:

* Ha van megoldás, akkor sem garantált a megtalálása.
* Ha nincs megoldás, akkor ezt véges állapottér-gráf esetén felismeri.

**Optimalitás**: Nem garantált az optimális célállapot (azaz az optimális megoldással elérhető célállapot) előállítása.

**Az egyes nem módosítható keresők abban különböznek egymástól, hogy hogyan választják meg az *o* operátort az *a* állapothoz.** Két megoldást említek meg:

1. **Próba-hiba módszer:** Véletlenszerűen választjuk meg *o*-t.
2. **Hegymászó módszer:** Azt az operátort választjuk, mely becslésünk szerint legközelebb visz a (valamelyik) célállapothoz

A nem módosítható keresők jelentőségét az adja, hogy újra lehet indítani őket. Ha az algoritmus zsákutcába fut, azaz az aktuális állapotra nincs alkalmazható operátor, akkor az algoritmust újra indítjuk (RESTART). Egyúttal úgy egészítjük ki a feladatot, hogy kizárjuk, hogy még egyszer ugyanabba a zsákutcába fussunk bele (ezt legegyszerűbben az ide vezető operátor alkalmazási előfeltételének kiegészítésével érhetjük el). Az újraindítások számát előre rögzítjük. Belátható, hogy az újraindítások számának növelésével nő a valószínűsége, hogy az algoritmus megoldást talál, feltéve, hogy van megoldás. Ha az újraindítások száma tart a végtelenhez, akkor a megoldás megtalálásának valószínűsége tart az 1-hez.

Az újraindítást alkalmazó nem módosítható megoldást kereső algoritmusokat restartos algoritmusoknak nevezzük.

A nem módosítható keresőket szokták egy hegyes, völgyes vidékre bepottyantott labdával is szemléltetni, amely mindig lefelé gurul, de egy kicsit pattog, mielőtt a lokális minimumban megállna. Ennek az felel meg, hogy a heurisztikánk azt az operátort választja, amely valamilyen szempontból kisebb értékű állapotba visz (lefelé gurulás), ha nincs ilyen, akkor véletlenszerűen választ egy operátort (pattogás), amíg ki nem derül, hogy mindig ugyanoda gurul vissza a labda. Ez a lokális minimum lesz.

A restart ebben a példában annak felel meg, hogy ha már találtunk egy lokális minimumot, akkor újra bedobjuk a labdát egy véletlen helyen.

Restartos módszernél a megtalált legkisebb lokális minimumot fogadjuk el a globális minimum közelítésének. Ez a közelítés annál pontosabb, minél nagyobb az újraindítások száma.

A restartos nem módosítható algoritmusoknak nagy jelentőségük van például a SAT probléma megoldásában. Ezt használják az úgynevezett random walk SAT megoldó algoritmusok.

**Nem módosítható gráfkereső algoritmusok**

* próba-hiba módszer (trial-and-error method)
  + adatbázis
    - aktuális csúcs
  + inicializálás
    - aktuális csúcs a ’start’ csúcs
    - a ’start’ csúcs tartalmazza a kezdő állapotot
  + ciklus
    - az aktuális csúcsra alkalmazok egy alkalmazható operátort, az így előálló új csúcs legyen az aktuális csúcs
    - azaz: akt=o(akt), ahol o egy alkalmazható operátor az akt-ra
    - a ciklusban egyet lépek előre
    - gond, hogy vakon lépek egyet előre, véletlenszerűen választok egy operátort => lehet, hogy rossz irányba megyek
  + pozitív kilépési feltétel
    - ha az aktuális csúcs terminális csúcs, azaz az aktuális csúcsban lévő állapot célállapot, akkor meg van a megoldás
    - a megoldás a ciklusban feltárt csúcsok sorrendjében adódik
  + negatív kilépési feltétel
    - ha a keresésre szánt idő eltelik vagy zsákutcába jutok, akkor feladom a keresést
  + technikai művelet
    - nincs, mert ez egy nem módosítható gráfkereső algoritmus (ez minden nem módosítható gráfkereső algoritmusra igaz)
* 3 kérdés
  + Ha van megoldás megtalálja?
    - valószínűtlen, azaz nem
    - nagyon nagy szerencse kell hozzá, hogy megtalálja
  + Ha nincs megoldás felismeri?
    - nem, mert csak szisztematikus keresők képesek erre
  + Ha talál megoldást, az optimális?
    - nagyon valószínűtlen, azaz nem
    - nincs garancia az optimális megoldásra
* ez egy „3 nemes” algoritmus
* nincs különösebb értelme az algoritmusnak, a gyakorlatban nem nagyon használjuk
* elméleti jelentősége -> minden más algoritmus felfogható ennek a javításaként

Legegyszerűbb, semmire se jó. Minden más ennek a javítása. Mint fentebb említettük, a próba-hiba módszer esetén az aktuális csúcsra egy véletlenszerűen kiválasztott alkalmazható operátort alkalmazunk.

**Adatbázis**: Az aktuális csúcs.

**Inicializálás**: A start csúcs legyen egyenlő a kezdő állapottal. S = k. És az aktuális csúcs legyen egyenlő a start csúccsal. AKT = S.

**Ciklus**: Első lépés, kiíratom az aktuális csúcsot, azért ha találunk megoldást tudjuk. Második lépés az aktuális csúcsra alkalmazok egy alkalmazható operátort. Vakon választom az operátort. Az így előálló új állapotból új csúcsot készítek. Harmadik lépés AKT = új csúcs. Akkor nem tudom folytatni, ha zsákutcába kerülök.

**Pozitív kilépési feltétel:** ha az aktuális csúcs terminális, akkor megtaláltuk a megoldást. A megoldás adódik az eddig kiírt csúcsok sorrendjében.

**Negatív kilépési feltétel:** Ha az aktuális csúcs zsákutca, azaz nincs alkalmazható operátor, akkor kiírom, hogy nem találtam megoldást.

**Technikai művelet:** Nincs.

Tulajdonságai:

Három kérdés:

* Ha van megoldás megtalálja? Nem valószínű.
* Ha nincs megoldás, felismeri? Nem, mert csak olyan algoritmus képes erre, ami szisztematikus és van benne körfigyelés.
* Ha talál megoldást, az optimális? Nem valószínű.

Ez az algoritmus egy nem-nem-nem algoritmus. Sose használjuk gyakorlatban. Minden más algoritmus ezt gondolja át. Rossz algoritmus.

Teljesség:

* Ha van megoldás, azt nem mindig találja meg.
* Ha nincs megoldás, akkor ezt véges állapottér-gráf esetén felismeri.

Optimalitás: nem garantált az optimális megoldás előállítása.

Véletlen választás (egyetlen) előnye: a végtelen ciklus szinte lehetetlen.

ÖTLET:

* Ha zsákutcába kerülünk, akkor restart.
* Kizárjuk, hogy még egyszer zsákutcába kerüljünk, megjegyezzük a csúcsot (bővítjük az adatbázist).

**Algoritmus javítása**

* 2 mód a javításra:
  + heurisztika segítségével
  + újraindítás segítségével

A heurisztika több lehetséges döntés közül kiválaszt egyet, anélkül, hogy a döntések pontos hatását ismernénk, általában gyorsabbnak kell, mintha az összes döntést kipróbálnám.

1. Legjobb, ha konstans idejű, azaz nincs benne, ’for’ ciklus, azaz O(1).
2. Legjobb, ha logaritmikus, azaz egy fát használ a döntéshez, a fa minden egyes csúcsánál eldönti, hogy jobbra vagy balra menjen, azaz O(logn).
3. Legjobb, azaz csak egy ’for’ ciklus van benne, vagy ha több van, akkor azok egymás után vannak, nincs olyan, hogy ’for’ cikluson belül ’for’ ciklus, azaz O(n).

Ami még elfogadható az az O(n \* logn), ami már elfogadhatatlan az az O(n \* n).

Gyakorlatban akkor mondjuk, hogy jó a heurisztika, ha lineáris.

Tipikus heurisztikák:

* 2 város távolsága közúton
  + heurisztika -> légvonalban mért távolság
* utasok összsúlya
  + heurisztika -> egy utas kb. 80 kg, utasok száma \* 80 kg => összsúly

A heurisztika az valamilyen közelítő érték. Valós érték ~ heurisztika.

**Restartos próba-hiba módszer**

* adatbázis
  + az aktuális csúcs, a maximális újraindítások száma, az eddigi újraindítások száma
* inicializálás
  + aktuális csúcs a ’start’ csúcs
  + a ’start’ csúcs tartalmazza a kezdő állapotot
  + a maximális újraindítások száma egy alkalmasan választott nagy szám pl.: 1000
  + az eddigi újraindítások száma legyen 0
* ciklus
  + az aktuális csúcsra alkalmazok egy alkalmazható operátort, az így előálló új csúcs legyen az aktuális csúcs
  + azaz: akt=o(akt), ahol o egy alkalmazható operátor az akt-ra
  + a ciklusban egyet lépek előre
  + gond, hogy vakon lépek egyet előre, véletlenszerűen választok egy operátort => lehet, hogy rossz irányba megyek
* pozitív kilépési feltétel
  + ha az aktuális csúcs terminális csúcs, azaz az aktuális csúcsban lévő állapot célállapot, akkor meg van a megoldás
  + a megoldás a ciklusban feltárt csúcsok sorrendjében adódik
* negatív kilépési feltétel
  + ha zsákutcába jutottam, akkor nincs gond lásd: technikai művelet
  + ha az újraindítások száma egyenlő a maximális újraindítások számával, akkor feladom a keresést
* technikai művelet
  + újraindítás, ha zsákutcába jutottam, azaz nincs alkalmazható operátor az aktuális csúcsra, akkor újraindítom a keresést: aktuális csúcs legyen a ’start’ csúcs
  + az eddigi újraindítások számát megnövelem 1-el és ugrás a ciklusra
  + azaz: akt=S; újraindításokszáma++; go to ciklus;
* 3 kérdés
  + Van megoldás, megtalálja?
    - ha az újraindítások száma tart a végtelenhez, akkor ez a válasz tart az igenhez
    - azaz, ha minél többször indítom újra, annál valószínűbb, hogy egyszer szerencsém lesz
  + Ha nincs megoldás, felismeri?
    - nem, mert erre csak szisztematikus algoritmus képes
  + Ha talál megoldást, az optimális?
    - nem, mert erre nincs garancia

Mivel a restartos próba-hibában van technikai művelet, ezért ez szoros értelemben módosítható algoritmusnak számít, hiszen csak a módosítható algoritmusoknál van technikai művelet.

Eddig 3 algoritmust néztünk: próba-hiba, restartos próba-hiba, hegymászó algoritmus

Javítási lehetőség: Ha a próba-hiba zsákutcába fut, indítsuk újra a start csúcsból. Kezdjük tiszta lappal. Véletlenszerűen választ és ez a restartosnál jól jön. Maximális újraindítások számát kiszoktuk kötni.

Megjegyezhetem az eddig feltárt zsákutcákat.

Adatbázis: az aktuális csúcs, a megjegyzett zsákutcák, az újraindítások száma és a maximális újraindítások száma.

Vezérlő:

1. *Inicializálás*: Az aktuális csúcs legyen a startcsúcs, a megjegyzett zsákutcák listája legyen üres, az újraindítások száma 0.
2. *Ciklus*: Az aktuális csúcsra alkalmazok egy véletlenszerűen kiválasztott alkalmazható operátort. Az így kapott új állapotot megvizsgálom, hogy benne van-e az ismert zsákutcák listájában. Ha igen, akkor ugrás a ciklus elejére. Ha nem, akkor az aktuális csúcs legyen az új állapotból készített csúcs.
3. *Tesztelés*: Ha az aktuális csúcs terminális csúcs, akkor a megoldás a képernyőre kiírt adatokból következtethető vissza.
4. Ha az aktuális csúcsra nincs alkalmazható operátor, azaz ha az aktuális csúcs zsákutca:

* Ha még nem értük el a maximum újraindítások számát, a megtalált zsákutcát felvesszük az adatbázisba, növeljük az újraindítások számát eggyel, az aktuális csúcs legyen a startcsúcs és ugrás a ciklus elejére.
* Ha elértük a maximális újraindítások számát, akkor kiírjuk, hogy nem találtunk megoldást. Az algoritmus tulajdonságai

Tulajdonságai:

Három kérdés:

Ha van megoldás megtalálja? Ez a válasz tart az igenhez amennyiben az újraindítások száma tart a végtelenhez. Minél nagyobb az újraindítás száma annál nagyobb az esélye, hogy megtalálom a megoldást.

Ha nincs megoldás, felismeri? Nem.

Ha talál megoldást, optimális? Nem.

Ez az algoritmus majdnem igen – nem – nem. Akár eshet végtelen ciklusba is.

Szokták használni, ha kiegészítik heurisztikával -> Restartos hegymászó algoritmus.

**Teljesség**:

* Ha van megoldás, azt nem mindig találja meg.
* Minél nagyobb az újraindítások száma, annál valószínűbb, hogy megtaláljuk a megoldást. Ha az újraindítások száma tart a végtelenhez, akkor annak a valószínűsége, hogy találunk megoldást, tart az egyhez.
* Ha nincs megoldás, akkor azt felismeri.

**Optimalitás**: nem garantált az optimális megoldás előállítása. A próba-hiba algoritmusnak elméleti jelentősége van. A restartos változatot véletlen sétának (random walk) nevezzük. A konjunktív normálformák kielégíthetősége legpraktikusabban ezzel az algoritmussal vizsgáltható.

**Hegymászó algoritmus**

Próba-hiba módszer kiegészítve heurisztikával.

* adatbázis
  + aktuális csúcs
* inicializálás
  + aktuális csúcs a ’start’ csúcs
  + a ’start’ csúcs tartalmazza a kezdő állapotot
* ciklus
  + az aktuális csúcsra alkalmazom a legjobb heurisztikájú alkalmazható operátort
  + az így előálló új csúcs legyen az aktuális csúcs
  + azaz: o=heurisztika(alkalmazható operátorok); akt = o(akt)
  + tehát előrelépek egyet a legjobb operátor segítségével
* pozitív kilépési feltétel
  + ha az aktuális csúcs terminális csúcs, azaz az aktuális csúcsban lévő állapot célállapot, akkor meg van a megoldás
  + a megoldás a ciklusban feltárt csúcsok sorrendjében adódik
* negatív kilépési feltétel
  + ha a keresésre szánt idő eltelik vagy zsákutcába jutok, akkor feladom a keresést
* technikai művelet
  + nincs, mert ez egy nem módosítható gráfkereső algoritmus (ez minden nem módosítható gráfkereső algoritmusra igaz)
* 3 kérdés
  + Ha van megoldás, megtalálja?
    - Ha jó a heurisztika, akkor igen.
  + Ha nincs megoldás felismeri?
    - Nem, mert erre csak szisztematikus algoritmusok képesek
  + Ha talál megoldás az optimális?
    - Nem, mert erre nincs garancia.

Ez egy majdnem igen, nem-nem algoritmus. Ennek már van jelentősége, ha az emberi tapasztalatot le tudom szűrni heurisztikába, akkor tudom ezt használni, ha nem akkor nem tudjuk. Ez egy jó algoritmus, ha tudunk heurisztikát alkalmazni. Jó heurisztikát írni nagyon nehéz.

Miért mondhatjuk, hogy ez egy javítása a próba-hiba módszernek?

* Azért mondjuk ezt, mert nem véletlenszerűen választok, hanem heurisztika segítségével.
* Azért javítás, mert ebben már van egy talán igen.
* Általánosítás, mert a véletlenszerűen visszaadhatja azt amit a heurisztika, és javítás, mert nagyobb eséllyel találja meg a megoldást.

Look-ahead heurisztika, előre tekintő heurisztika: Megbecsüli minden operátorról, hogy mennyire visz közel a megoldáshoz. Vagy minden operátort kipróbál és minden operátor eredményét megbecsüli, hogy milyen közel van a megoldáshoz. Az előre tekintés lehet 1 mélységben, több mélységben. Több mélységben legenárolom az összes lehetséges gyermeket, majd azok összes lehetséges gyermekét. Heurisztikától azt várjuk el, hogy olcsólegyen. Look- ahead annál drágább minél nagyobb mélységbe tekintünk.

Annyiban különbözik, hogy nem vakon választ operátort, hanem egy heurisztika segítségével.

Leggyakrabban a look-aheadet használjuk. Az elképzelés, az hogy mászok felfelé, keresem a fogásokat és azt választom ami stabil. Magyarul a valamilyen szempontból a legjobb operátort fogom választani. Minden más tekintetbe megegyezik a próba-hibával.

Három kérdés:

Ha van megoldás megtalálja? Talán, minél jobba heurisztika annál valószínűbb.

Ha nincs megoldás, felismeri? Nem, mer csak akkor igen, ha szisztematikus, nincs bene körfigyelés.

Ha talál megoldást, optimális? Nem.

Ez az algoritmus egy talál-nem-nem algoritmus. Jobb, mint az előző. Ha van, olyan probléma, ahol lehet jó heurisztikát használni használjuk, egyébként nem használjuk.

**Restartos hegymászó algoritmus**

A véletlenszerűség helyett heurisztikával választok, azaz determinisztikusan választok.

A heurisztika alapvetően determinisztikus, tehát ugyanabban a helyzetben ugyanazon operátorok közül ugyanazt az operátort fogja választani.

Ha újraindítom, ugyanabba a zsákutcába kerül

* emiatt csak akkor lehet restartos hegymászó algoritmust csinálni
  + ha a heurisztikának van véletlen része is, azaz nem mindig a legjobbat választom
  + vagy, ha kikötöm, hogy ugyanabba a zsákutcába már nem lehet belefutni
* adatbázis
  + az aktuális csúcs az újraindítások maximális száma
  + az eddigi újraindítások száma
  + az eddig felderített zsákutcák halmaza
* inicializálás
  + S = k
  + akt = S
  + maxUjraindit = 1000;
  + ujraIndit = 0;
  + zsakutca = ureshalmaz;
* ciklus
  + az aktuális csúcsra alkalmazom a legjobb heurisztikájú alkalmazható operátort
  + az így előálló új csúcs legyen az aktuális csúcs
  + azaz: o=heurisztika(alkalmazható operátorok); akt = o(akt)
  + tehát előrelépek egyet a legjobb operátor segítségével
  + ha az operátor ismert zsákutcába vezet, akkor azt az operátort kiveszem az alkalmazható operátorok közül
* pozitív kilépési feltétel
  + ha az aktuális csúcs terminális csúcs, azaz az aktuális csúcsban lévő állapot célállapot, akkor meg van a megoldás
  + a megoldás a ciklusban feltárt csúcsok sorrendjében adódik
* negatív kilépési feltétel
  + ha az újraindítások száma elérte a maximális újraindítások számát, akkor feladom a keresést
* technikai művelet
  + újraindítás: feltétele, hogy az aktuális csúcs zsákutca, azaz nincs alkalmazható operátor. ebben az esetben az aktuális csúcsot felveszem a zsákutcák közé.
  + az újraindítások számát megnövelem 1-el.
  + az aktuális csúcs legyen a ’start’ csúcs.
  + ugrás a ciklusra
* 3 kérdés
  + Ha van megoldás, felismeri?
    - Igen, viszonylag kevés újraindítással is, ha jó a heurisztika.
  + Ha nincs megoldás, felismeri?
    - Nem, mert erre csak szisztematikus algoritmusok képesek
  + Ha talál megoldást, az optimális?
    - Nem, mert erre nincs garancia.
* Ennek az algoritmusnak van saját neve: véletlen séták (random walk)
  + jelentős, nagyon olcsó, kevés memóriát használ
  + a nem szisztematikus keresők közül az egyik legjobb.

Példa a zsákutcára

* vegyük a 4 királynő problémát
  + egy 4x4-es táblára kell feltenni a 4 királynőt, hogy ne üssék egymást
  + a lenti ábrán a 4. királynőt nem tudom feltenni, mert minden szabad mező ütésben van. Azaz, ez egy zsákutca. Ekkor újra kell indítanom a keresést.

A képen shoji látható

Automatikusan generált leírás

A hegymászó algoritmus nem heurisztikus. Ha ugyan arra problémára indítom, ugyan oda jut. Megoldás az, hogy megkell jegyezni az eddig feltárt zsákutcákat és nem szabad belelépni a már feltárt zsákutcábaS.

A heurisztikának adhatok véletlen részt is, ez is egy megoldás.

Random walk, véletlen séták: Azt nevezzük hogy restartos hegymászó úgy hogy megjegyzem a zsákutcákat. Ha nem feltétel, hogy szisztematikus keresőt kell használni akkor ez a legjobb választás.

A restartos hegymászó módszer megegyezik a hegymászó módszerrel annyi kiegészítéssel, hogy egy előre meghatározott számú újraindítást engedélyezünk. A hegymászó módszert újraindítjuk, ha az zsákutcába fut. Ha elértük a maximális újraindítási számot és zsákutcába futunk, akkor az algoritmus megáll, mert nem talált megoldást.

Fontos, hogy minden zsákutcából tanuljon az algoritmus, azaz ne tudjon kétszer ugyanabba a zsákutcába futni. Enélkül a heurisztika mindig ugyanabba a zsákutcába vezetné a hegymászót az újraindítás után, kivéve, ha a heurisztikának van véletlen része. A tanulás többféleképpen is történhet. Legegyszerűbb az állapottér-reprezentációt megváltoztatni, úgy, hogy a zsákutcába futáskor az aktuális állapotot töröljük az állapotok halmazából. Másik megoldás, hogy az adatbázist bővítjük egy tiltott állapotok listájával.

Két esetben érdemes használni:

1. ha tanul, azaz megjegyzi a felderített zsákutcákat
2. ha a heurisztika nem determinisztikus

Mért lógnak ki a restartos algoritmusoka nem módosíthatók közül? Mert a technikai művelet az újraindítás.

Restartos helyett újra indíthatónak is lehet hívni.