**Backtrack és javításai**

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 adatszerkezetet 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 operátorok 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?

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

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

**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.

## Módosítható gráfkereső algoritmusok

* Visszalépéses keresés: Technikai művelet: visszalépés (backtrack).
  + Sima backtrack
  + Mélységi kvadrátos backtrack
  + Körfigyeléses backtrack
  + Ág-korlátos backtrack
* Keresőfával keresők: Technikai művelet: kiterjesztés (extension).

## Visszalépéses keresők

A módosítható megoldáskereső algoritmusok egy fajtája a visszalépéses (vagy idegen szóval: backtrack) kereső, melynek több változata is létezik. A backtrack keresők alapötlete: ne csak az aktuális csúcsot tároljuk az adatbázisban, hanem azokat a csúcsokat is, melyeken keresztül az aktuális csúcsba eljutottunk. Ez azt jelenti tulajdonképpen, hogy az adatbázisban az állapottérgráfnak most már nagyobb részét fogjuk tárolni: a startcsúcsból az aktuális csúcsba vezető utat.

A backtrack keresők nagy előnye, hogy a keresés nem kerülhet zsákutcába. Ha az aktuális csúcsból nincs továbblépés a gráfban, akkor visszalépünk az aktuális csúcs szülőjébe, és abból próbálunk ezúttal más irányba lépni. Ez a speciális lépés – amit visszalépésnek neveznek – adta a nevét ennek a fajta keresőnek.

Logikus, hogy minden egyes az adatbázisban tárolt csúcsban az állapot mellett azt is el kell tárolni, hogy a csúcsból eddig merrefelé próbáltunk továbblépni. Vagyis: minden csúcsban regisztrálni kell azokat az operátorokat, melyeket a csúcsban tárolt állapotra nem próbáltunk még alkalmazni. Abban a pillanatban, mikor egy operátort alkalmaztam az állapotra, az operátort törlöm a csúcsban tárolt regisztrációból.

**Backtrack**

* Alap ötlet: ha zsákutcába futunk, akkor 1-et visszalépek és hogy ne lehessen még egyszer ugyanabba a zsákutcába visszafutni, minden operátor alkalmazás után az adott operátort kitörlöm a csúcsból. Nem az összes operátort törlöm ki, csak abból az egy csúcsból.

**Alap backtrack**

1. adatbázis
   1. az aktuális út, amely a ’start’ csúcsból vezet az aktuális csúcsba
2. inicializálás
   1. a ’start’ legyek a következő struktúra
   2. tartalmazza a kezdőállapotot,
   3. a ’start’ csúcs mélységét, ami 0
   4. a ’start’ csúcs szülőcsúcsát, ami 0, azaz nincs
   5. a kezdőállapotra alkalmazható operátorok halmazát
   6. azaz S = (k, 0, null, {o | k-ra alkalmazható operátorok}), azaz a csúcsban tárolom az állapotot, a csúcs mélységét, a csúcs szülőjét és a csúcsban lévő állapotra alkalmazható operátorok halmazát
   7. Akt = S, azaz innen indítom a keresést
3. ciklus
   1. az aktuális csúcsból véletlenszerűen választok egy alkalmazható operátort, legyen ez a ’o’ operátor.
   2. Ezt kitörlöm az alkalmazható operátorok közül, de csak az aktuális csúcsból.
   3. Az Aktuális cúcsban lévő állapotra alkalmazom a ’o’ operátort, így létrejön egy új állapot.
   4. Az új állapotba létrehozom az új csúcsot, a következő módon:
      1. Új = (új, Akt.melyseg + 1, Akt, {o | az új állapotra alkalmazható operátor}), azaz az Új csúcs szülőcsúcsa: az aktuális csúcs (Akt)
      2. az aktuális csúcs legyen az új csúcs
      3. azaz Akt = Új
4. pozitív kilépési feltétel
   1. ha az aktuális csúcs terminális csúcs, azaz aktuális csúcsban lévő állapot célállapot, akkor megvan a megoldás, a megoldás adódik a szülő csúcs referencián visszafelé haladva a ’start’ csúcsig
5. negatív kilépési feltétel
   1. nem az, hogy zsákutcába futok, hiszen a zsákutcából ki tudok jönni visszalépéssel
   2. a visszalépés hatására visszajutok az előző csúcsba, ahol egyel kevesebb operátor lesz, hiszen az előbb használt operátort kitöröltem
   3. elképzelhető, hogy már nem maradt alkalmazható operátor, ilyenkor újból vissza kell lépni
   4. ha maradt operátor, akkor azt kipróbálom, de elképzelhető, hogy az is zsákutcába visz
   5. legrosszabb esetben vissza kell lépnem egészen a ’start’ csúcsig
   6. a ’start’ csúcsból már nem lehet visszalépni, hiszen annak már nincs szülőcsúcsa
   7. a negatív kilépési feltétel az, hogy a ’start’ csúcsból kellene visszalépni, azaz az aktuális csúcs a ’start’ csúcs, és már nincs egy alkalmazható operátor sem
6. technikai művelet
   1. egy technikai művelet, a visszalépés
      1. feltétel: zsákutcába lépjünk, az aktuális csúcsba ne legyen egy alkalmazható operátor sem
      2. ilyenkor visszalépek, a visszalépés kódja
         1. Akt = Akt.szülő, azaz visszalépek a szülőcsúcsba
         2. ha vermet használok, akkor a visszalépés módja az, hogy kiveszem a verem tetején lévő dolgot Pop-pal
         3. ha rekurzívan valósítom meg a visszalépéses keresést, akkor a visszalépés kódja a return

* 3 kérdés
  + Ha van megoldás, megtalálja?
    - csak abban az esetben, ha a gráf körmentes, mert ha van a gráfban kör, akkor van arra esély, hogy végtelen sokszor pörög a körben, azaz végtelen ciklusba jutok
    - ha nincs ilyen, akkor a válasz igen
  + Ha nincs megoldás, felismeri?
    - igen, mert szisztematikus, de csak abban az esetben, ha a gráf körmentes
    - ha a gráf nem körmentes, akkor végtelen ciklusba eshet
  + Ha talál megoldást, az optimális?
    - nem, erre nincs garancia
* körmentes gráf esetén, ez egy igen-igen-nem algoritmus
* jelentős, gyakran használjuk
* kicsi a memóriaigénye, hiszen csak az aktuális utat, azaz egyetlen utat tárolok le
* viszonylag kevés olyan probléma van, ami a körmentes, ezért szükséges a javítása
* 2 féle javítás
  + mélységi korlátos backtrack
    - alapötlet: csak egy bizonyos mélységig engedem futni a backtracket, ha elérem a mélységi korlátot, akkor visszalépek
  + körfigyeléses backtrack
    - alapötlet: ha eljutok ugyanabban az állapotba, amiben már jártam, az azt jelenti, hogy egy körben pörgök
    - ilyenkor inkább visszalépek, és azzal kizárom azt, hogy egy körben pörögjek

Felépítés:

Jegyzet

|  |
| --- |
| **Adatbázis**: az állapottér-gráfban a startcsúcsból az aktuális csúcsba vezető út.  **Műveletek**:   * Operátorok: az állapottér-reprezentációban adottak. * Visszalépés: technikai művelet, mely az adatbázisban tárolt út legalsó csúcsának a törlését jelenti.   **Vezérlő**: Az adatbázist inicializálja, az aktuális csúcsra műveletet hajt végre, teszteli a célfeltételt, majd ez alapján eldönti, hogy leáll-e a kereséssel, vagy pedig újra megvizsgálja az aktuális csúcsot. Részletesen a vezérlő működése:   1. *Inicializálás*: A startcsúcsot egyedüli csúcsként elhelyezi az adatbázisban. startcsúcs= kezdőállapot + az összes operátor regisztrálva 2. *Ciklus*: 3. Ha az adatbázis üres, leáll a keresés. Nem találtunk megoldást. 4. Az adatbázisban tárolt út legalján elhelyezkedő (azaz az adatbázisba legkésőbb berakott) csúcsot kiválasztjuk; ezt aktuális csúcsnak fogjuk nevezni. Jelöljük az aktuális csúcsban tárolt állapotot a -val! 5. Tesztelés: Ha az a célállapot, akkor leáll a keresés. A megtalált megoldás: maga az adatbázis (mint út). 6. Megvizsgálja, hogy van-e olyan operátor, melyet még nem próbáltunk alkalmazni az a -ra. Azaz: van-e még az aktuális csúcsban operátor regisztrálva?  * Ha van ilyen, jelöljük el o -val! Töröljük o -t az aktuális csúcsból. Teszteljük az o alkalmazási előfeltételét az a -ra. Ha az teljesül, akkor alkalmazzuk az o -t az a -ra, és az így kapott o (a) állapotot beszúrjuk az adatbázisban tárolt út aljára. Az új csúcsban az o (a) mellett az összes operátort regisztráljuk. * Ha nincs ilyen, akkor a vezérlő visszalép. |

Saját jegyzet

**Adatbázis**: az aktuális, azaz a start csúcsból az aktuális csúcsba vezető út.

**Inicializálás:**

Matematikai jelölés: (Rendezett négyes ahol az első dolog, k az állapot, szám, ami a mélység, itt 0)

**S = (k, 0, NULL, {….})** ahol a csúcs típus így néz ki: // {…} az alkalmazható operátorok

Class Csúcs

{

Állapot A;

int mélység;

csúcs szülő;

List<operator> AlkOp;

}

Csúcs S = new Csúcs();

S.A = k; // A start csúcs állapota a kezdő állapot

S.mélység = 0;

S.szülő = null;

S.AlkOp = alkalmazható operátorok halmaza. A k-ra alkalmazható operátorok halmaza.A rendezett 4-ben {..} jelöli.

Inicializálás 2 lépésből áll: Léptre hozom a start csúcsot. Második: az aktuális csúcs legyen a start csúcs.

**S = (k, 0, NULL, {….})**

**AKT = S;**

**Ciklus**

*Első lépés:* Az aktuális csúcsban tárolt operátorok közül, valamilyen heurisztika segítségével választok egyet.

*Második lépés:* Ezt az operátort törlöm az aktuális csúcsban tárolt alkalmazható operátorok közül.

*Harmadik lépés:* Ezt az operátort alkalmazom az aktuális csúcsban tárolt állapotra, így előáll egy új állapot.

*Negyedik lépés:* Az új állapotból létrehozok egy új csúcsot, úgy hogy:

Új csúcs = (állapot, mélység, szülőcsúcs, alkalmazható operátorok)

Új csúcs = ( új állapot, AKT.mélyseg + 1 // az új csúcs egyel lejjebb lesz, mint a szülőcsúcsa, AKT, Az új állapotra alkamazható operátorok halmaza)

*Ötödik lépés:* AKT = új csúcs

Mitől áll meg ez a ciklus?

**Pozitív kilépési feltétel:** Ha az aktuális csúcs terminális, akkor megvan a megoldás. A megoldás adódik a szülő referenciánkon visszafelé haladva a start csúcsig.

**Negatív kilépési feltétel:** Ha a start csúcsból kellene visszalépni, azaz amikor a start csúcs válik zsákutcává, akkor kiírom, hogy nincs megoldás.

**Technikai művelet:** A visszalépés. Ha az aktuális csúcs zsákutca, azaz az alkalmazható operátorok halmaza üres, akkor visszalépek, ami azt jelenti, hogy az én szülőcsúcsom lesz az aktuális. Az én az AKT-ot jelenti. Azaz **AKT = AKT.szülő**. Ez abban az esetben van így ha nem rekurzívan csináltam, ha ciklussal csináltam. Ciklussal: return false; vagy return 0; Valahogy visszakell térnem sikertelenül. Visszatérek egy szintel feljebb és ott folytatom a keresést. Rekurzívan könnyebb leprogramozni, de lassabban fut.

Csak akkor szabad iteratívan, ha nyelvben van garbage collector. (ha már nincs referencia egy objektumra, akkor azt az objektumot fellehet szabadítani.) Ha nincs GC a nyelvnek akkor 3 utasítás kell hozzá:

Hogy kell visszalépni, ha nyelv nem tartalmaz Garbage collectort:

1. temp = AKT;
2. AKT = AKT.szülő
3. free(temp);

free(): felszabadítja a dinamikus memóriát.

Három kérdés:

Ha van megoldás megtalálja? Nem mert nincs benne körfigyelés. végtelen ciklusba eshet. Viszont, ha a gráf körmentes, akkor ha van megoldás megtalálja.

Ha nincs megoldás felismeri? Nem mert végtelen ciklusba eshet a körök miatt. Viszont ha agráf körmentes, akkor a válasz igen.

Ha talál megoldás, optimális? Nem valószínű, azaz nem.

Körmentes gráfok esetén ez egy igen – igen – nem algoritmus.

Nem ritka, h a gráf körmentes, iyle szempontból jó algoritmus. Úgyhogy javítsuk: mélységkorláttal és kör figyeléssel.

Tulajdonságai:

**Teljesség**:

* Ha van megoldás, akkor véges állapottér-gráfban megtalálja azt.
* Ha nincs megoldás, akkor ezt véges állapottér-gráf esetén felismeri.

**Optimalitás**: nem garantált az optimális megoldás előállítása.

**Mélységi korlátos backtrack** –backtrack with depth limit

1. adatbázis:
   1. az aktuális út, amely a ’start’ csúcsból vezet az aktuális csúcsba
   2. van egy plusz változóm, a mélységi korlát, ami egy alkalmasan választott kicsi szám
   3. általába 10-30 mélységig működőképes, utána már nagyon lassú
2. inicializálás
   1. a ’start’ legyek a következő struktúra
   2. tartalmazza a kezdőállapotot,
   3. a ’start’ csúcs mélységét, ami 0
   4. a ’start’ csúcs szülőcsúcsát, ami 0, azaz nincs
   5. a kezdőállapotra alkalmazható operátorok halmazát
   6. azaz S = (k, 0, null, {o | k-ra alkalmazható operátorok}), azaz a csúcsban tárolom az állapotot, a csúcs mélységét, a csúcs szülőjét és a csúcsban lévő állapotra alkalmazható operátorok halmazát
   7. Akt = S, azaz innen indítom a keresést
   8. a mélységi korlát, ami egy kicsi szám
3. ciklus
   1. az aktuális csúcsból véletlenszerűen választok egy alkalmazható operátort, legyen ez a ’o’ operátor.
   2. Ezt kitörlöm az alkalmazható operátorok közül, de csak az aktuális csúcsból.
   3. Az Aktuális cúcsban lévő állapotra alkalmazom a ’o’ operátort, így létrejön egy új állapot.
   4. Az új állapotba létrehozom az új csúcsot, a következő módon:
      1. Új = (új, Akt.melyseg + 1, Akt, {o | az új állapotra alkalmazható operátor}), azaz az Új csúcs szülőcsúcsa: az aktuális csúcs (Akt)
      2. az aktuális csúcs legyen az új csúcs
      3. azaz Akt = Új
4. pozitív kilépési feltétel
   1. ha az aktuális csúcs terminális csúcs, azaz aktuális csúcsban lévő állapot célállapot, akkor megvan a megoldás, a megoldás adódik a szülő csúcs referencián visszafelé haladva a ’start’ csúcsig
5. negatív kilépési feltétel
   1. nem az, hogy zsákutcába futok, hiszen a zsákutcából ki tudok jönni visszalépéssel
   2. a visszalépés hatására visszajutok az előző csúcsba, ahol egyel kevesebb operátor lesz, hiszen az előbb használt operátort kitöröltem
   3. elképzelhető, hogy már nem maradt alkalmazható operátor, ilyenkor újból vissza kell lépni
   4. ha maradt operátor, akkor azt kipróbálom, de elképzelhető, hogy az is zsákutcába visz
   5. legrosszabb esetben vissza kell lépnem egészen a ’start’ csúcsig
   6. a ’start’ csúcsból már nem lehet visszalépni, hiszen annak már nincs szülőcsúcsa
   7. a negatív kilépési feltétel az, hogy a ’start’ csúcsból kellene visszalépni, azaz az aktuális csúcs a ’start’ csúcs, és már nincs egy alkalmazható operátor sem
6. technikai művelet
   1. egy technikai művelet, a visszalépés
      1. feltétel: zsákutcába lépjünk, az aktuális csúcsba ne legyen egy alkalmazható operátor sem
      2. ilyenkor visszalépek, a visszalépés kódja
         1. Akt = Akt.szülő, azaz visszalépek a szülőcsúcsba
         2. ha vermet használok, akkor a visszalépés módja az, hogy kiveszem a verem tetején lévő dolgot Pop-pal
         3. ha rekurzívan valósítom meg a visszalépéses keresést, akkor a visszalépés kódja a return
   2. ha **az aktuális csúcs mélysége megegyezik a mélységi korláttal**, vagy az aktuális csúcs zsákutca, akkor visszalépek, azaz Akt = Akt.Szülő

* 3 kérdés
  + Ha van megoldás, megtalálja?
    - Igen, de csak akkor, ha a mélységi korlát felett van.
    - ezért kell alkalmasan választani mélységi korlátot, hiszen, ha rosszul választom meg, akkor nem találok megoldást
    - minél kisebb a szám, annál gyorsabb az algoritmus
  + Ha nincs megoldás, felismeri?
    - igen, de csak azt, hogy a mélységi korlát felett nincs megoldás
  + Ha talál megoldást, az optimális?
    - Nem, erre nincs garancia.
* ez egy igen-igen-nem algoritmus, de csak akkor, ha jól választottam meg a mélységi korlátot
* jelentős, gyakran használjuk, egyszerű (nem kellenek plusz ciklusok)
* kevés a memóriaigénye

**Adatbázis**: Ugyan az, mint sima backtrack esetén kiegészítve a mélységi korláttal.

**Inicializálás**: Ugyan az, mint sima backtrack esetén, azzal kiegészítve, hogy a mélységi korlátot beállítom egy megfelelően választott konstanssal. Mondjuk mélységikorlát = 15.

**Ciklus**: Teljesen ugyan az, mint sima backtrack

**Negatív és pozitív kilépési feltétel:** Ugyan az.

**Technikai** **művelet**: Visszalépés, ha az aktuális csúcs zsákutca vagy az aktuális csúcs mélysége nagyobb egyenlő, mint a mélységi korlát, akkor visszalépés, azaz AKT = AKT.szülő. Ez azt jelenti, hogy a mélységi korlát alá nem engedem be.

Három kérdés:

Ha van megoldás, megtalálja? Nem, csak akkor, ha mélységi korlát fölött van, akkor igen.

Ha nincs megoldás, felismeri? Nem, mert csak azt ismeri fel, ha mélységi korlát felett nincs megoldás.

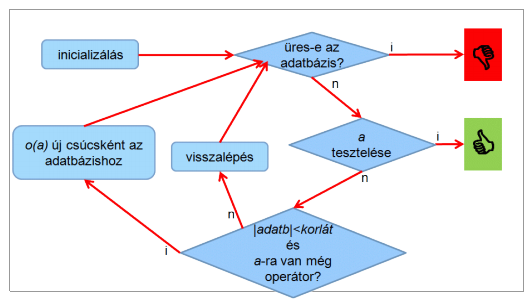
Ha talál megoldás optimális? Nem valószínű.

Ha jól megválasztottam a mélységi korlátot, akkor ez egy nagyon jó algoritmus IGEN-NEM-NEM

Játszani szoktak vele: 25 mélységi korlát, utána 20, 15 stb.

Olcsó algoritmus, mert a sima backtrackhez képest csak egy plusz if van benne.

Úgy működik hogy nem bánja, ha futok a körbe, ez nem engedi h végtelen sokszor végig fussak a körön mert elérem a mélységi korlátot és visszalépek.



**Kör figyeléses backtrack** – Circle detection backtrack

1. adatbázis
   1. az aktuális út, amely a ’start’ csúcsból vezet az aktuális csúcsba
2. inicializálás
   1. a ’start’ legyek a következő struktúra
   2. tartalmazza a kezdőállapotot,
   3. a ’start’ csúcs mélységét, ami 0
   4. a ’start’ csúcs szülőcsúcsát, ami 0, azaz nincs
   5. a kezdőállapotra alkalmazható operátorok halmazát
   6. azaz S = (k, 0, null, {o | k-ra alkalmazható operátorok}), azaz a csúcsban tárolom az állapotot, a csúcs mélységét, a csúcs szülőjét és a csúcsban lévő állapotra alkalmazható operátorok halmazát
   7. Akt = S, azaz innen indítom a keresést
3. ciklus
   1. az aktuális csúcsból véletlenszerűen választok egy alkalmazható operátort, legyen ez a ’o’ operátor.
   2. Ezt kitörlöm az alkalmazható operátorok közül, de csak az aktuális csúcsból.
   3. Az Aktuális cúcsban lévő állapotra alkalmazom a ’o’ operátort, így létrejön egy új állapot.
   4. Az új állapotba létrehozom az új csúcsot, a következő módon:
      1. Új = (új, Akt.melyseg + 1, Akt, {o | az új állapotra alkalmazható operátor}), azaz az Új csúcs szülőcsúcsa: az aktuális csúcs (Akt)
      2. az aktuális csúcs legyen az új csúcs
      3. azaz Akt = Új
4. pozitív kilépési feltétel
   1. ha az aktuális csúcs terminális csúcs, azaz aktuális csúcsban lévő állapot célállapot, akkor megvan a megoldás, a megoldás adódik a szülő csúcs referencián visszafelé haladva a ’start’ csúcsig
5. negatív kilépési feltétel
   1. nem az, hogy zsákutcába futok, hiszen a zsákutcából ki tudok jönni visszalépéssel
   2. a visszalépés hatására visszajutok az előző csúcsba, ahol egyel kevesebb operátor lesz, hiszen az előbb használt operátort kitöröltem
   3. elképzelhető, hogy már nem maradt alkalmazható operátor, ilyenkor újból vissza kell lépni
   4. ha maradt operátor, akkor azt kipróbálom, de elképzelhető, hogy az is zsákutcába visz
   5. legrosszabb esetben vissza kell lépnem egészen a ’start’ csúcsig
   6. a ’start’ csúcsból már nem lehet visszalépni, hiszen annak már nincs szülőcsúcsa
   7. a negatív kilépési feltétel az, hogy a ’start’ csúcsból kellene visszalépni, azaz az aktuális csúcs a ’start’ csúcs, és már nincs egy alkalmazható operátor sem
6. technikai művelet
   1. csak a visszalépés feltétele lesz más
   2. ha az aktuális csúcsban lévő állapt megtalálható bármely előtte lévő csúcsban vagy az aktuális csúcs zsákutca, akkor visszalépés
   3. Akt = Akt.Szülő

* 3 kérdés
  + Ha van megoldás, megtalálja?
    - igen
  + Ha nincs megoldás, felismeri?
    - igen
  + Ha talál megoldást, optimális?
    - nem, erre nincs garancia
* ez egy tisztán igen-igen-nem algoritmus
* ez egy jó algoritmus, csak a mélységi korlátos backtrackhez képes van benne egy plusz cikus
* de legalább nem kell hozzá plusz memória, mert csak az előző csúcsokban kell keresnem, hogy ott megtalálható-e az aktuális állapot
  + - 3 szerzetes, 3 kannibál:

(3,0,3,0,” bal”)

1. S = ((3,0,3,0,” bal”), 0, null, {o01, o02, o11}), választom az ’o11’-t
2. S = ((3,0,3,0,” bal”), 0, null, {o01, o02})
   1. A = ((2,1,2,1,” jobb”), 1, S, {o11, 010}), választom az ’o10’-t
3. S = ((3,0,3,0,” bal”), 0, null, {o01, o02})
   1. A = ((2,1,2,1,” jobb”), 1, S, {o11})
   2. B = ((3,0,2,1,” bal”), 2, A, {o10, o01, o02}), választom ’o02’-t
4. S = ((3,0,3,0,” bal”), 0, null, {o01, o02})
   1. A = ((2,1,2,1,” jobb”), 1, S, {o11})
   2. B = ((3,0,2,1,” bal”), 2, A, {o10, o01})
   3. C = ((3, 0, 0, 3, „jobb”), 3, B, {o10, o02}) választom „o02” -t
5. S = ((3,0,3,0,” bal”), 0, null, {o01, o02})
   1. A = ((2,1,2,1,” jobb”), 1, S, {o11})
   2. B = ((3,0,2,1,” bal”), 2, A, {o10, o01})
   3. C = ((3, 0, 0, 3, „jobb”), 3, B, {o10})
   4. Akt = D = ((3,0,2,1,” bal”) , 4, C, {o01,o02,o10}), , innen vissza fog lépni a körfigyeléses, mert D állapota megegyezik a B állapotával, visszalépés után ez lesz a memóriában:
6. S = ((3,0,3,0,” bal”), 0, null, {o01, o02})
   1. A = ((2,1,2,1,” jobb”), 1, S, {o11})
   2. B = ((3,0,2,1,” bal”), 2, A, {o10, o01})
   3. Akt = C = ((3,0,0,3,” jobb”), 3, B, {o10}), itt csak egy op-ot tudok választani, azzal fogok továbblépni.

* Ha nincs GC (garbage collector), akkor a visszalépés kódja 3 utasításból áll:
  + Temp = Akt;
  + Akt = Akt.Szülő;
  + Free (Temp); //azaz felszabadítom azt a memóriát, amire a Temp mutat

**Adatbázis**: Ugyan az, mint sima backtrack.

**Inicializálás**: Ugyan az, mint sima backtrack esetén.

**Ciklus**: Teljesen ugyan az, mint sima backtrack.

**Negatív és pozitív kilépési feltétel:** Ugyan az.

**Technikai művelet:** Visszalépés. Ha az aktuális csúcs zsákutca vagy az aktuális csúcs állapota már szerepelt valamely előző csúcsban, akkor visszalépés, azaz AKT = AKT.Szülő.

Itt egy plusz while ciklus van. Kör figyelő algoritmus:

bool Körfigyelés()

{

Csúcs TEMP = AKT.szülő

while (temp != null) {

if(akt.A.equals(temp.A)) return true;

else temp = temp.szulo

}

return false;

}

Drága, hiszen az eddigi legbelső ciklusban van egy ciklus. Ha cikluson belül kell még egy ciklust futtatnom az drága. De legalább nem kell plusz memória hozzá, hanem elég az aktuális memória. Viszont ez nem engedi meg h akár egyszer is végig menjek egy körön, amikor végig mennék, akkor kidob. Ez drága, de hatékony.

Három kérdés:

Ha van megoldás, megtalálja? IGEN mert ez egy szisztematikus algoritmus, és van benne körfigyelés.

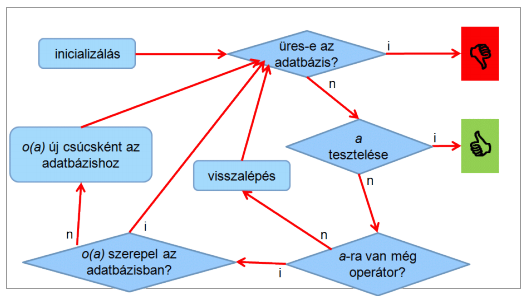
Ha nincs megoldás, felismeri? IGEN mert szisztematikus és van benne kör figyelés

SZISZTEMATIKUS: legrosszabb esetben feltárja az egész gráfot. (Ha nincs megoldás) kipróbálja az össze lehetséges alkalmazható operátort.

Ha talál megoldás, optimális? Nem valószínű.

Ez az első IGEN-IGEN-NEM algoritmus.

Egyetlen különbség van e között és a sima backtrack között: a technikai műveletben van egy VAGY.



## Mélységi korlátos, körfigyeléses backtrack

**Adatbázis**: Ugyan az, mint sima backtrack.

**Inicializálás**: Ugyan az, mint sima backtrack esetén.

**Ciklus**: Teljesen ugyan az, mint sima backtrack.

**Negatív és pozitív kilépési feltétel:** Ugyan az.

**Technikai művelet:** Visszalépés. Ha az aktuális csúcs zsákutca vagy az aktuális csúcs állapota megtalálom valamely előző csúcs állapotában, vagy az aktuális csúcs mélysége nagyobb egyenlő, mint a mélységi korlát, akkor visszalépés, azaz AKT = AKT.Szülő.

IGEN- IGEN- NEM

Jó algoritmus, olcsó (nem drágább, mint a körfigyeléses) egy plusz if van benne. Ahhoz képest, hogy jobb, mint a körfigyeléses, nem drágább.

Ismétlés:

* Az előző órán az alap backtracket és két javítását néztük
  + azért kell javítani az alap backtracket, mert végtelen ciklusba eshet
  + ha a gráfban van kör, akkor újra és újra körbe járhat, emiatt valahogyan ki kell zárni, hogy végtelen sokszor körbe menjen
    - 2 módszer:
      * mélységi korlátos backtrack => véges sokszor pöröghet a körben, előbb-utóbb kilép a körből
        + ehhez nem kellett plusz ciklus, mert az algoritmus egyszerű és gyors
      * körfigyeléses backtrack => egyszerűen megnézem, hogy ugyanebben az állapotban jártam-e már
        + így biztosan nem fog a körben feleslegesen pörögni, viszont ehhez plusz ciklus kell
        + ez egy igen-igen-nem algoritmus, szóval ez egy jó algoritmus
        + ahhoz, hogy 3 igenes algoritmus legyen, ehhez kell a következő javítás