

## *A Puzzle kirakójáték megoldása*

*- prolog gyakorlat -*

Adott a **3x3**-as puzzle kirakójáték. A feladat a következő:

1. Jelenítsük meg grafikusán az állapotokat és a közöttük – esetleg – fennálló kapcsolatokat. Javasolt a gráfok használata.
2. Ábrázoljuk a játékot, azaz határozzunk meg egy állapotteret, ahol tároljuk a lehetséges állapotokat. Jelöljük meg a *kitüntetett* elemet, azaz az üres kockának a helyét.
3. Kódoljuk a lépéseket (javasolt a lépések kódolása az üres kockához viszonyítva).
4. Írjuk fel a lehetséges lépések sorozatát. Azaz: melyik állapotból melyikbe lehet lépni és kódoljuk az illető lépést.
5. Írjunk egy algoritmust, amely megkeresi két állapot között a lehetséges utat (út=lépések sorozata). Ügyeljünk arra, hogy kerüljük el a végtelen ciklusokat, melyek a gráf ciklusainak bejárásából adódnak.

### **Megoldás:**

#### **1. Megjelenítés**

## 2. Ábrázolás

A játék kódolására egy lehetőség a **3x3**-as négyzet **9** hosszúságú vektorban való tárolása. A vektor elemei az 1-9 közötti számok, ahol a *kilences az üres hely*.

Egy másik lehetőség a nullától nyolcig való kódolás, ebben az esetben a nullás az üres hely... a megoldások ekvivalensek.

Ily módon a cél, hogy egy adott kezdeti konfigurációból (amit a program futásakor határozunk meg) eljussunk az **[1, 2, 3, 4, 5, 6, 7, 8, 9]** végállapotba.

## 3. A lépések kódolása

Érvényes lépés a **9**-es elemnek a permutációja egy szomszédjával. Kell tehát kódolni a **szomszéd** és a **lépés** fogalmát.

Két lehetőségünk van a lépések kódolásánál. Az egyik – könnyebb – amikor *csak* a 3x3-as esetre írjuk fel a megoldást. Ebben az esetben adott a játék konfigurációja; csak a lépéseket fogjuk specifikálni:

A felsorolás logikája az, hogy **minden** pozícióból – a bal felső sarokból kezdődően – leírjuk a lehetséges lépéseket. A **kilences** szám mozgását kódoljuk: amikor lefele „megy”, akkor a mozgás második koordinátája egy, amikor jobbra, akkor az első. A felsorolás **mindig** a még le nem írt mozgásokat tartalmazza, tehát mindig csak jobbra illetve lefele lehet „lépni”. Figyeljük meg, hogy a kilencesen kívül **csak** változókat találunk a predikátumok felépítésében.

%A11

lepes ([**9**, A12, A13, A21, A22, A23, A31, A32, A33], [A12, **9**, A13, A21, A22, A23, A31, A32, A33], (1, 0)).  
lepes ([A12, **9**, A13, A21, A22, A23, A31, A32, A33], [**9**, A12, A13, A21, A22, A23, A31, A32, A33], (-1, 0)).  
lepes ([**9**, A12, A13, A21, A22, A23, A31, A32, A33], [A21, A12, A13, **9**, A22, A23, A31, A32, A33], (0, 1)).  
lepes ([A21, A12, A13, **9**, A22, A23, A31, A32, A33], [**9**, A12, A13, A21, A22, A23, A31, A32, A33], (0, -1)).

%A12

lepes ([A11, **9**, A13, A21, A22, A23, A31, A32, A33], [A11, A13, **9**, A21, A22, A23, A31, A32, A33], (1, 0)).  
lepes ([A11, A13, **9**, A21, A22, A23, A31, A32, A33], [A11, **9**, A13, A21, A22, A23, A31, A32, A33], (-1, 0)).  
lepes ([A11, **9**, A13, A21, A22, A23, A31, A32, A33], [A11, A22, A13, A21, **9**, A23, A31, A32, A33], (0, 1)).  
lepes ([A11, A22, A13, A21, **9**, A23, A31, A32, A33], [A11, **9**, A13, A21, A22, A23, A31, A32, A33], (0, -1)).

%A13

lepes ([A11, A12, **9**, A21, A22, A23, A31, A32, A33], [A11, A12, A23, A21, A22, **9**, A31, A32, A33], (0, 1)).  
lepes ([A11, A12, A23, A21, A22, **9**, A31, A32, A33], [A11, A12, **9**, A21, A22, A23, A31, A32, A33], (0, -1)).

%A21

lepes ([A11, A12, A13, **9**, A22, A23, A31, A32, A33], [A11, A12, A13, A22, **9**, A23, A31, A32, A33], (1, 0)).  
lepes ([A11, A12, A13, A22, **9**, A23, A31, A32, A33], [A11, A12, A13, **9**, A22, A23, A31, A32, A33], (-1, 0)).  
lepes ([A11, A12, A13, **9**, A22, A23, A31, A32, A33], [A11, A12, A13, A31, A22, A23, **9**, A32, A33], (0, 1)).  
lepes ([A11, A12, A13, A31, A22, A23, **9**, A32, A33], [A11, A12, A13, **9**, A22, A23, A31, A32, A33], (0, -1)).

%A22

lepes ([A11, A12, A13, A21, **9**, A23, A31, A32, A33], [A11, A12, A13, A21, A23, **9**, A31, A32, A33], (1, 0)).  
lepes ([A11, A12, A13, A21, A23, **9**, A31, A32, A33], [A11, A12, A13, A21, **9**, A23, A31, A32, A33], (-1, 0)).  
lepes ([A11, A12, A13, A21, **9**, A23, A31, A32, A33], [A11, A12, A13, A21, A32, A23, A31, **9**, A33], (0, 1)).  
lepes ([A11, A12, A13, A21, A32, A23, A31, **9**, A33], [A11, A12, A13, A21, **9**, A23, A31, A32, A33], (0, -1)).

%A23

lepes ([A11, A12, A13, A21, A22, **9**, A31, A32, A33], [A11, A12, A13, A21, A22, A33, A31, A32, **9**], (0, 1)).  
lepes ([A11, A12, A13, A21, A22, A33, A31, A32, **9**], [A11, A12, A13, A21, A22, **9**, A31, A32, A33], (0, -1)).

%A31

lepes ([A11, A12, A13, A21, A22, A23, **9**, A32, A33], [A11, A12, A13, A21, A22, A23, A32, **9**, A33], (1, 0)).  
lepes ([A11, A12, A13, A21, A22, A23, A32, **9**, A33], [A11, A12, A13, A21, A22, A23, **9**, A32, A33], (-1, 0)).

%A32

lepes ([A11, A12, A13, A21, A22, A23, A31, **9**, A33], [A11, A12, A13, A21, A22, A23, A31, A33, **9**], (1, 0)).  
lepes ([A11, A12, A13, A21, A22, A23, A31, A33, **9**], [A11, A12, A13, A21, A22, A23, A31, **9**, A33], (-1, 0)).

A másik lehetőség a specifikációra a puzzle játék elemzése és a lépések specifikációjának az automatizálása a következők szerint:

1. megkeressük a kilences pozícióját a listában. Ez megtehető az **nth1** paranccsal – lásd **help(nth1)**.
2. kiszámítjuk a pozíció függvényében a sor-, illetve oszlop-számot; legyen ez **(i, j)**.
3. figyelembe véve a **puzzle** játék méreteit, ez esetben **3x3**, kiszámítjuk a lehetséges lépéseket. Itt azt nézzük, hogy az oszlop-szerinti harmadik pozícióból ne lehessen jobbra lépni, stb...

A fenti algoritmus – amellet, hogy rövidebb, biztosít egy általánosítást a **puzzle** méretei szerint – a „harmadik” kicserélésével lehet bármely méretű **puzzle**-re alkalmazni. (feladat)

#### 4. Utak keresése

Jelöljük egy állapotot **P1**-gyel és legyen **P2** a vég-állapot (mi lesz ez?). A cél, hogy egy utat megtaláljunk a P1-ből a P2-be, jelöljük az utat szintén egy listával, ebben az esetben a lista elemei **lépések** lesznek – ezek a **lépes** predikátum harmadik argumentumai. Felírhatjuk, hogy

**út(P1,P2,[Egylépes]) :- lépes(P1,P2,Egylépes).**

ami azt a tényt fejezi ki, hogy a **P1**-ből a **P2** állapotba el tudunk jutni – és pedig avval az úttal, mely egyetlen lépésből áll – ha a **P1**-ből a **P2**-be van lépés, amely lépést az **Egylépes** változóban tároljuk. Az egy lépésnél hosszabb utakat rekurzívan adjuk meg:

**út(P1,P2,[Egylépes|Tobbi]) :- lépes(P1,P3,Egylépes),út(P3,P2,Tobbi).**

ahol a **P3** egy közbülső állapot, melyet a rendszer keres – a visszalépéses algoritmus segítségével.

A fenti algoritmus-implementáció annyiban hibás, hogy nem mondjuk meg, milyen mélységig lehet kiterjeszteni a puzzle-gráfot.

#### 5. Működő algoritmus implementálása

A gráf mélységének megállapítására szükségünk van egy változóra, legyen a neve **Szint**. A fenti algoritmust tehát ki kell egészíteni úgy, hogy egy adott **Max\_Szint**-nél tovább ne keressen állapotokat.

Végezzük el a kiegészítést (használjuk az SWI-prolog rendszert):

```
% predikátum argumentumok:
% L1=ahonnan, L2=ahova, Lista=mozgassorozat,
% LSzam=_megtett_ lepesek szama = melyseg,
% MaxL=legnagyobb melyseg

ut(L1,L2,[M],_,_):- lépes(L1,L2,M).
ut(L1,L2,[EgyL|Tobbi],M,MaxH) :-
    M<MaxH,
    M1 is M+1,
    lépes(L1,L3,EgyL),
    ut(L3,L2,Tobbi,M1,MaxH).

% masodik megoldas - a bejart utak tarolasa
% predikátum argumentumok:
% Honnan=ahonnan, Hova=ahova,
Ut=mozgassorozat,
% LSzam=melyseg, MaxHossz=legnagyobb melyseg
% Bejart= a mar meglátogatott állapotok

% altalanosan:
% ut(Honnan,Hova,Ut,MaxHossz,Halmaz)

ut(L1,L2,[M],_,_):- lépes(L1,L2,M).
ut(L1,L2,[EgyL|Tobbi],MH,Halmaz) :-
    lépes(L1,L3,EgyL),
    not(memberchk(L3,Halmaz)),
    length(Halmaz,N),
    N < MH,
    append(Halmaz,[L3],NottHalmaz),
    ut(L3,L2,Tobbi,MH,NottHalmaz).

% a megoldast megtalalo predikatum
% L-kezdoadallapot, Út-lepesek, MH-max.Hossz
puzz(L,Ut,MH):-
    ut(L,[1,2,3,4,5,6,7,8,9],Ut,MH,[L]).
```

(figyeljük meg, hogy a fenti kód – a lépések keresése – független a játék méretétől)

## Prolog segédanyag:

- A **prolog** predikátumokkal operál, a célja az, hogy egy **tu-dásbázis** bevitelére nyomán a rendszernek feltett kérdésekre válaszolni tudjon. A predikátumok tartalmazhatnak **atomokat** vagy **változókat**. Minden kisbetűvel kezdődő név atom, a számok is atomok, és minden nagybetűvel vagy az aláhúzással '\_' kezdődő név – azonosító – változóként van kezelve (első labor/szeminárium).

### % Atomok:

mária, teréz, ádám, születésnap, 14, 1984.

### % Változók:

Név, Dátum, Összeg, Lista

- Megkülönböztetünk tényeket és szabályokat. A tények **statikus** „információk” egy-egy objektum – atom – állapotáról illetve avval kapcsolatban. A **szabályok** dinamikus komponensei a prolog rendszernek: megmondják, hogy hogyan lehet **származtatni** új tényeket a már meglévőkből. Általában az adatok közötti kapcsolatok egy kompakt leírását teszik lehetővé.

### % Tények:

**gyereke**(jános,ádám). **gyereke**(jános,éva).

**gyereke**(istván,jános). **gyereke**(istván,emese).

**gyereke**(pali,istván). **gyereke**(pali,zsuzsa).

### % Szabályok:

**őse**(X,Y) :- **gyereke**(Y,X).

**őse**(X.Y) :- **gyereke**(Z.X).**őse**(Z.Y).

A mellékelt példában **nem** kell leírni, hogy őse Ádám Jánosnak, és szintén nem kell megmondani, hogy Ádám Palinak is őse a vázolt – „mini” – genealógiai adatbázisban. A rekurzív specifikáció egy erőssége a prolog rendszernek.

(figyeljük meg az **operátorok** használatát: a vessző „,” az és-t, a „:-” az implikációt – „**akkor ha**” relációt. A „**vagy**” relációt a szabályok egymás alá írásával tudjuk implementálni.)

- A prologban – a tények beolvasása után – a rendszert kérdezzük – az SQL-hez hasonlóan. Megkérdezhetjük egy kijelentés igazságértékét, pl. azt, hogy őse-e Éva Istvánnak: a válasz, hogy IGEN, őse.

?- őse(éva,istván).

Yes.

- Hasonlóan megkérdezhetjük a rendszert arról, hogy **kik** egy személy ősei. Értelemszerűen megjelenik, hogy minden adatbázisban szereplő személy őse Palinak.

?- őse(X,pali).

istván,zsuzsa,jános,emese,ádám,éva, No.

- Listák prologban. A megoldások sok alkalommal – látható fent is – nem egyediek, szeretnénk például azokat egy listában tárolni további feldolgozás érdekében. Ezt a **findall** predikátummal lehet elérni, lásd a példában.

?- findall(X,őse(X,jános),Ősök).

Ősök=[ádám,éva]

Az **Ősök** egy lista, melynek elemei atomok.

- **Műveletek listákkal.**

Listák predikátumokban megjelenhetnek struktúrájuk szerint. Ebben az esetben zárójelek - [ , , , ... ] – határolják a listát, és egy függőleges vonal választja el az elemenkénti felsorolást a lista maradékától.

Ha például **Ősök**=[ádám,éva,istván,emese], akkor a lista a következő formákban is felírható:

**Ősök** = [ádám | [éva,istván,emese]]

**Ősök** = [ádám,éva | [istván,emese]]

**Ősök** = [ádám | [éva | [istván | [emese]]]]

A strukturált felírás a listák elemzésénél fontos, amikor egyik vagy másik komponens helyére változót teszünk.

**Ősök** = [Első,Második | Maradék].

eredménye

Első = ádám

Második = éva

Maradék = [istván, emese]

Sok megírt függvényt találunk a listákkal történő műveletekre. Néhány közülük:

- **length(Lista, N)** – a **List**a listának a hosszát teszi az **N**-be
- **append(L1, L2, Ered)** – akkor igaz, ha az **L1** és **L2** listák egyesítése az **L3**-at adja. A predikátum argumentumainak bármelyike lehet változó.
- **member(E, Lista)** – akkor igaz, ha a **List**a egy eleme megfeleltethető az **E**-nek. A predikátum használata hasonló az **append** parancséhoz.
- **select(E, Lista, Marad)** – igaz, ha az **E** elem és **Marad** listák egyesítésének az eredménye a **List**a. Változó lehet bármelyik argumentum.
- További műveletek kereshetőek a prolog **help(függvény)** . parancsával. Javasolt, hogy írjunk egy függvénynevet, amelyet már ismerünk, majd keressünk a függvényhez tartozó csoportban.

```
append(L1,L2,[1,2,3]).  
eredményei  
L1=[], L2=[1,2,3]  
...  
L1=[1,2,3], L2=[]
```

```
select(alma,[körte,alma,szilva],Marad).  
eredménye  
Marad = [körte,szilva]  
select(E,[1,2,3,4],Marad).  
eredménye négy (E,Marad) páros.
```

### A prolog működése:

A prolog a „**backtracking**” algoritmust – visszalépéses keresést – valósítja meg. A rendszer a lekérdezéskor – a „**?-**” prompt-nál – megadott kérdést próbálja kifejtetni a beolvasott tudásbázisára támaszkodva. Amennyiben **változókat** talál, a rendszer megkeresi az illeszthető változókat, majd azon változókkal keres a tudásbázisban. A szabályokat mindig kiterjeszti az adott sorrend szerint, egészen addig, amíg megtalálja a szabályokat – ez esetben sikeres a levezetés, a válasz **Yes** –, vagy nem tud többet lépni, ez esetben meg sikertelen, a válasz **No**.  
(próbáljuk ki a fentieket az SWI-prolog rendszerében.)

Segédfeladatok:

1. találjuk meg egy lista hosszát;
2. adjuk össze egy lista elemeit;
3. ellenőrizzük, hogy egy lista számtani sorozat-e.