16. Problēmu risināšanas piemēri

Nodaļas saturs

Nodaļā ievietotie attēli, programmu piemēri un tabulas

Jebkuras netriviālas problēmas risināšanai parasti ir vairāk nekā viens iespējamais variants, un katrs šāds risināšanas process sastāv vismaz no 2 soļiem:

- problēmas risināšanā iesaistīto datu reprezentācija,
- algoritmu izvēle vai izveide.

Ja otrs no punktiem nerada šaubas, tad datu attēlošanas variantu izskatīšana bieži vien netikai ņemta vērā pietiekoši nopietni.

Šajā nodaļā tiks apskatīti dažādi algoritmi, dažiem no kuriem datu attēlošanas principa izvēle atstāj būtisku iespaidu uz risinājumu.

16.1. Kārtošanas algoritmi

Kārtošanas algoritms sakārto vērtību virkni tā, ka diviem elementiem – tas, kas ir vairāk pa kreisi, ir mazāks vai vienāds par to, kas ir vairāk pa labi. Tādējādi starp jebkuriem diviem elementiem, kas var parādīties sarakstā, jābūt definētai attiecībai mazāks (<) vai lielāks (>).

Nākošie piemēri tiks demonstrēti uz skaitļu masīviem un tiks izmantots salīdzināšanas operators (<).

16.1.1. Burbuļkārtošana (bubble-sort)

Burbuļkārtošana ir primitīvākais kārtošanas veids, kas notiek šādi:

- saraksta apstrāde notiek pa vairākiem piegājieniem (ne vairāk kā N-1 piegājieni),
- katrā piegājienā iet cauri sarakstam un salīdzina blakusesošos elementus: ja kreisais ir lielāks par labo, tad tos apmaina vietām,
- ja piegājiena laikā nenotiek neviena apmaiņa, tad saraksts ir sakārtots.

Tradicionālajās programmēšanas valodās šis algoritms ir ļoti vienkārši realizējams, tomēr deklaratīvajā manierē tas izskatās nedaudz ķēpīgi:

```
bubblesort([],[]):-!.
bubblesort(X,Z):-shuffle(X,Y,N), N>0, !, bubblesort(Y,Z); Z=X.
shuffle([A],[A],0):-!.
shuffle([A,B|X],[B|Z],N):-A>B, !, Y=[A|X], shuffle(Y,Z,M), N is
    M+1.
shuffle([A|X],[A|Y],N):-shuffle(X,Y,N).
```

Pirmais noteikums apskata speciālgadījumuj, kad saraksts ir tukšs.

Otrais noteikums veic vienu sajaukšanu ar apmaiņu uzskaiti (shuffle/3) un, ja izmaiņas ir bijušas (N>0), tad izsauc nākamo piegājienu, citādi nosaka, ka saraksts ir sakārtots.

Ceturtais noteikums izskatās "visnesaprotamākais", bet nozīmē tikai elementu apmainīšanu vietām, ja kreisais lielāks par labo, un piegājiena turpināšanu saraksta astei.

```
?- bubblesort([4,1,7,3],X).

X = [1,3,4,7]

yes
```

16.1.2. Kārtošana ar sapludināšanu (merge-sort)

Kārtošana ar sapludināšanu ir labāka 2 aspektos:

- tas ir teorētiski labāks algoritms (strādā ar ātrumu $O(n \cdot log\ n)$),
- tas ir labāk piemērots loģiskās programmēšanas valodām.

Kārtošana ar sapludināšanu strādā pēc būtības rekursīvi, un katrs algoritma solis ietver 2 fāzes:

- sašķelšana – sadala māsīvu divās (jebkādās) daļās,
- divus sakārtotus masīvus sapludina vienā.

Spēja sakārtot balstās uz to, ka saraksts tiek sadalīts līdz sastāvdaļām garumā 1, kas automātiski nozīmē sakārtotību, bet pēc tam tās tiek sapludinātas kopā.

Sašķelšanu organizē, pamīšus liekot vienu elementu vienā, otru otrā sarakstā:

```
?- splitlist([a,b,c],X,Y).

X = [a,c]
Y = [b]
```

```
yes
?- splitlist([a,b,c,d],X,Y).

X = [a,c]
Y = [b,d]

yes
```

splitlist/3 realizācija:

```
splitlist([],[],[]):-!.
splitlist([A],[A],[]):-!.
splitlist([A,B|X],[A|Y],[B|Z]):-splitlist(X,Y,Z).
```

mergelist/3 divus sakārtotus sarakstus sapludina par vienu sakārtotu:

```
?- mergelist([1,3,6,7],[2,4,5,8],X).

X = [1,2,3,4,5,6,7,8]

yes
```

mergelist/3 realizācija:

```
mergelist([],[],[]):-!.
mergelist([],A,A):-!.
mergelist(A,[],A):-!.
mergelist([A|X],[B|Y],[A|Z]):-A<B, !, mergelist(X,[B|Y],Z).
mergelist(X,[B|Y],[B|Z]):-mergelist(X,Y,Z).
```

mergesort/2 apvieno abus iepriekšējos predikātus, realizējot sakārtošanu:

```
?- mergesort([3,7,1,8,4],X).

X = [1,3,4,7,8]

yes
```

mergesort/2 realizācija:

```
mergesort([],[]):-!.
mergesort([A],[A]):-!.
mergesort(X,Y):-splitlist(X,X1,X2), mergesort(X1,Y1),
    mergesort(X2,Y2), mergelist(Y1,Y2,Y).
```

16.2. Sakārtotas kopas reprezentēšana ar bināru meklēšanas koku

16.2.1. Binārs meklēšanas koks

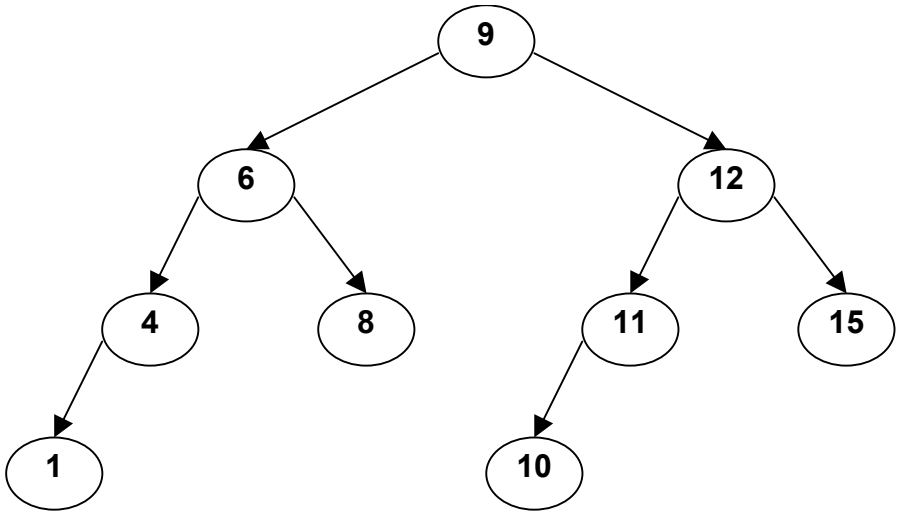
Elementa sameklēšana sakārtotā sarakstā, izmantojot Prolog konstrukcijas, nedod priekšrocības pret nesakārtotu sarakstu – tāpat praktiski ir jāpārstaigā viss saraksts (ātrdarbība O(n)). Alternatīvs risinājums ir binārs meklēšanas koks.

Binārs koks (*binary tree*) ir struktūra, kas var būt tukša vai sastāvēt no viena vai vairākiem elementiem – mezgliem (*nodes*), no kuriem viens elements ir galvenais – sakne (*root*). Katram elementam var būt divi bērni (*children*) – kreisais un labais (*left child*, *right child*).

Binārā meklēšanas kokā (*binary search tree*) ir papildus noteikts, ka kreisā bērna atslēga (*key*) ir mazāka par paša mezgla atslēgu, bet labā – lielākā. Šādi sakārtotā kokā, ja tas ir pietiekoši balansēts (šeit: pietiekoši sazarots), mezglu pēc atslēgas var atrast ļoti ātri, ideāli balansētā – ar ātrumu *O(log n)*.

Koks (tāpat kā Prolog saraksts) ir rekursīva struktūra, tāpēc apakšstruktūra, kas sākas no jebkura mezgla, arī ir koks ar doto mezglu saknes lomā. Ja ņem vērā, ka tukšs koks (mezgls) arī ir koks, tad nākošajā attēlā ir ieraugāmi kopā 19 koki (9 koki no katra no mezgliem + 10 tukšās "bērnu" vietas, tukšie koki, pie dažādiem elementiem). Nav grūti aprēķināt, ka apakškoku skaits jebkurā binārā kokā ir N*2+1, kur N – mezglu skaits.

Att. 16-1. Binära meklēšanas koka piemērs.



16.2.2. Binära koka reprezentēšana valodā Prolog

Koks sastāv no mezgliem, un katrs mezgls ir vai un tukšs (t.i., viņa nav), vai arī sastāv no (vismaz) 3 elementiem:

- kreisais bērns,
- atslēga,
- labais bērns.

Valodā Prolog ir iespējami vairāki varianti mezgla atveidošanai.

Pirmais variants – izmantojot struktūru ar funktoru:

- tukša mezgla apzīmēšanai ieviest noteiktu vērtību, piemēram, *nil*,
- netukša mezgla apzīmēšanai ieviest 3-vietīgu struktūru ar funktoru, piemēram, *node*.

Šajā variantā apakškoki, ar saknēm mezglos 4, 8 un 12 izskatās šādi:

```
node(node(nil,1,nil),4,nil)
node(nil,8,nil)
node(node(node(nil,10,nil),11,nil),12,node(nil,15,nil))
```

Pirmais variants – izmantojot sarakstu:

- tukšu mezglu apzīmē tukšs saraksts,
- netukša mezgla apzīmēšanai izmanto sarakstu garumā 3.

Tie paši apakškoki reprezentācijā ar sarakstu.

```
[[[],1,[]],4,[]]
[[],8,[]]
[[[],10,[]],11,[]],12,[[],15,[]]]
```

Nākošajos piemēros tiks izmantots variants ar sarakstiem. Pilns koks (Att. 16-1), reprezentēts ar sarakstiem, izskatās šādi:

```
[[[[],1,[]],4,[]],6,[[],8,[]]],9,[[[[],10,[]],11,[]],12,[[],15,[]]]]
```

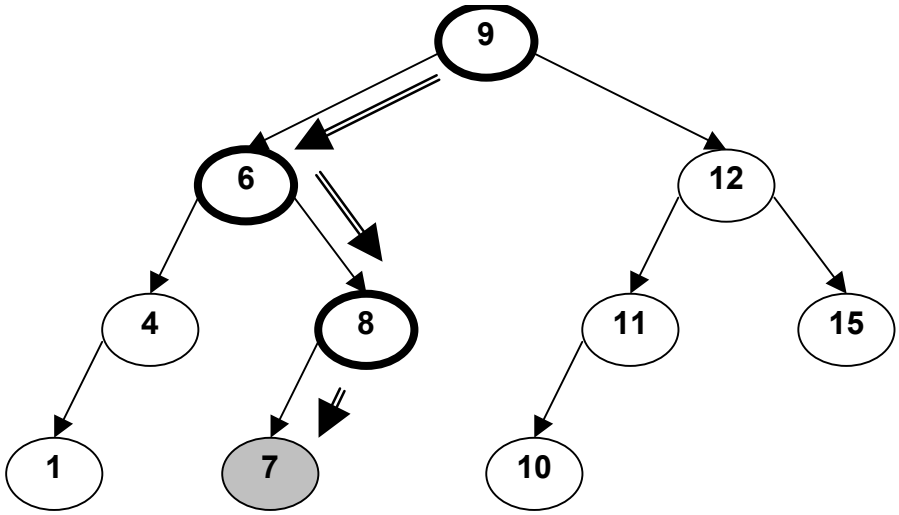
16.2.3. Mezgla pievienošana

Pēc mezgla pievienošanas bināram meklēšanas kokam joprojām jāatbilst prasībām par vecāku un bērnu atslēgu attiecībām.

Jaunas vērtības pievienošana bināram meklēšanas kokam sākas ar sakni un katrā solī (pie katra mezgla) notiek vienā no variantiem:

- ja mezgls tukšs – pieliek jauno vērtību un procesu beidz,
- ka mezgla atslēga sakrīt ar jauno vērtību – nepieliek un procesu beidz,
- ja jaunā vērtība mazāka par mezgla atslēgu – turpina procesu kreisajā bērnā,
- citādi (ja jaunā vērtība lielāka par mezgla atslēgu) – turpina procesu labajā bērnā.

Att. 16-2. Mezgla pievienošanas piemērs.



Balstoties uz iepriekš definētiem 4 gadījumiem, arī ir veicama mezgla pievienošanas predikāta *addnode/3*, izveide (argumenti: vecais koks, pievienojamā vērtība, koks pēc pievienošanas):

```
addnode([],B,[[],B,[]]):-!.
addnode([X,B,Y],B,[X,B,Y]):-!.
addnode([X,B,Y],A,[Z,B,Y]):-A<B, !, addnode(X,A,Z).
```

```
addnode([X,B,Y],C,[X,B,Z]):-addnode(Y,C,Z).
```

Lai iegūtu noteiktu koku, izmantojot *addnode/3*, mezgli ir jāpievieno noteiktā, pareizā secībā. Tālākais piemērs parāda *addnode/3* izmantošanu vienkāršam piemēram:

```
?- addnode([],9,X),addnode(X,6,Y),addnode(Y,12,Z).

X = [[],9,[]]
Y = [[[],6,[]],9,[]]
Z = [[[],6,[]],9,[[],12,[]]]

yes
```

Ērtākai koka ievadīšanai būtu ērta koka izveidošana no saraksta – tiek padots saraksts ar pievienojamo elementu vērtībām pareizā secībā *addnodes/2*, kas izmanto apakšpredikātu *addnodes/3*, jo koka nulles punkts ir izpildes sākumā, bet saraksta – beigās, tādēļ vajadzīgs papildus arguments "koka veidošanai", lai, iztukšojoties sarakstam, to nosūtītu uz izeju:

```
addnodes([],T,T).
addnodes([A|X],R,T):-addnode(R,A,S), addnodes(X,S,T).
addnodes(L,T):-addnodes(L,[],T).
```

Koka izveidošana (Att. 16-1) un atsevišķi nodalītu mezgla ar vērtību 7 pievienošana (kura varēja tikt pievienota arī saraksta beigās) (Att. 16-2) notiek šādi:

```
?- addnodes([9,6,12,4,8,11,15,1,10],S), addnode(S,7,T).

S = [[[[[],1,[]],4,[]],6,[[],8,[]]],9,[[[[],10,[]],
    11,[]],12,[[],15,[]]]]
T = [[[[[],1,[]],4,[]],6,[[[],7,[]],8,[]]],9,[[[[],10,[]],
    11,[]],12,[[],15,[]]]]

yes
```

16.2.4. Koka pārstaigāšana

Pārstaigāšana (*traversal*) ir operācija pār datu struktūru, kuras laikā katrs struktūras elements tiek apmeklēts tieši vienu reizi, un ar to tiek veikta noteikta darbība (piemēram, izdrukāšana).

Bināram kokam izšķir 3 tipiskos rekursīvās pārstaigāšanas veidus:

- *preorder* – vispirms pašu mezglu, tad kreiso un labo apakškoku,
- *inorder* – vispirms kreiso apakškoku, tad pašu mezglu, tad labo apakškoku,
- *postorder* – vispirms apakškokus, tad pašu mezglu.

Par piemēru tiks apskatīts *inorder* apstaigāšanas veids, kas binära meklēšanas koka gadījumā nodrošina pārstaigāšanu alfabētiskā secībā.

Koka pārstaigāšanas rekursīvie algoritmi ir ļoti viegli realizējami. Predikāts *printree/1* izdrukā koka vērtības *inorder* secībā, starp vērtībām liekot tukšumus:

```
printtree([]):-!.
printtree([L,K,R]):-printtree(L),write(K),tab(1),printtree(R).
```

Darbībā:

```
?- addnodes([9,6,12,4,8,11,15,1,10,7],T),printtree(T).
1 4 6 7 8 9 10 11 12 15

T = [[[[[],1,[]],4,[]],6,[[[],7,[]],8,[]]],9,[[[[],10,[]],
    11,[]],12,[[],15,[]]]]

yes
```

16.3. Problēmu risināšana ar meklēšanu stāvokļu telpā

Meklēšana stāvokļu telpā (*state space search*) ir problēmu risināšanas stratēģija, kad noteiktā secībā tiek pārstaigātas problēmas konfigurācijas (stāvokļi), lai beigās nonāktu pie tāda stāvokļa, kas atbilst noteiktam kritērijam.

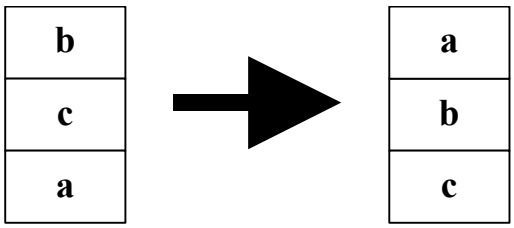
Vispārīgā gadījumā iespējamo stāvokļu skaits var būt bezgalīgs vai arī ļoti liels, tāpēc praktiski meklēšana notiek noteiktā stāvokļu apakšgrafā.

Meklēšana stāvokļu telpā tiks apskatīta uz vienkārša piemēra bāzes – klucīšu sakraušana pareizā secībā (3 klucīšu gadījums).

16.3.1. Problēmas apraksts

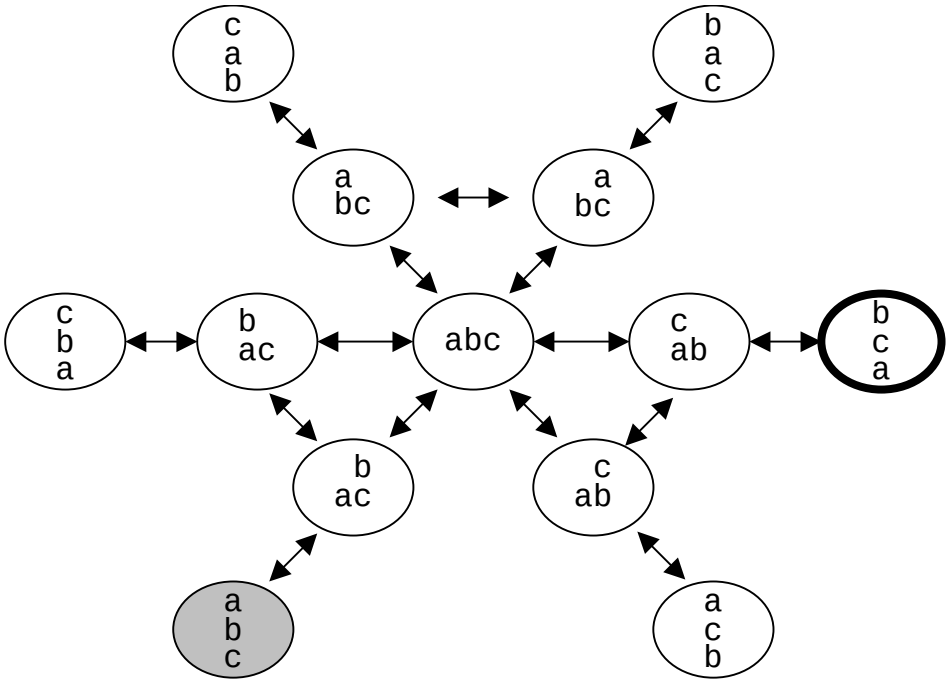
Doti 3 klucīši: a,b,c, kas ir sakrauti kolonnā viens uz otra brīvi izvēlētā secībā, piemēram [b,c,a]. Veicot primitīvas pārkraušanas darbības, panākt, lai tiktu izveidota kolonna, kurā klucīši būtu sakrauti pareizā secībā: [a,b,c]: drīkst kraut klucīšus no kolonnas uz zemi un otrādi, kā arī starp kolonnām, no kolonnas drīkst noņemt klucīti tikai tad, ja tas ir kolonnas augšpusē.

Att. 16-3. Klucīšu pārlikšanas problēmas piemērs.



Visi iespējamie stāvokļi pārlikšanas procesā kopā ar iespējamajiem gājieniem (t.i., saitēm starp stāvokļiem) veido orientētu grafu:

Att. 16-4. Klucīšu pārlikšanas problēmas iespējamo stāvokļu grafs (iezīmējot stāvokļus no Att. 16-3).



16.3.2. Problēmas reprezentācija Prolog struktūrās

Viens no variantiem ir vienu stāvokli reprezentēt ar 3 sarakstiem, kur katrs saraksts nozīmē vienu kolonnu, bet katra kolonna arī ir saraksts, kura galva atbilst kolonnas augšai.

Tādējādi problēmas sākuma stāvokli (atbilstoši Att. 16-3) var apzīmēt kā:

```
[[b,c,a],[],[]]
```

Bet beigu nosacījumam atbilstu veseli 3 stāvokļi:

```
[[a,b,c],[],[]]
[[],[a,b,c],[]]
[[],[],[a,b,c]]
```

Tas var tikt apzīmēts kā 3 Prolog noteikumi:

```
cubesok([[a,b,c],[],[]]).
cubesok([[],[a,b,c],[]]).
cubesok([[],[],[a,b,c]]).
```

16.3.3. Stāvokļa maiņa

Stāvokļa maiņa nozīmē paņemt klucīti no vienas kolonnas augšas un uzlikt otrai kolonnai. Lai nebūtu jāuzskaita visi iespējamie varianti, kā no kolonnas uz kolonnu var pārlikt klucīti, der šāds algoritms:

- izvēlēties kolonnu nr. 1, kurā ir vismaz viens klucītis,
- izvēlēties kolonnu nr. 2,
- pārlikt klucīti no kolonnas 1 uz kolonnu 2.

Kolonnas izvēlei ļoti ērts ir jau iepriekš izmantotais predikāts *removeelement/3*, kurš izmet no saraksta vienu elementu un nodrošina visu iespējamo variantu pārstaigāšanu. Ar tā palīdzību var uzbūvēt predikātu *move/2*:

```
removeelement([A|X],A,X).
removeelement([B|X],A,[B|Y]):-removeelement(X,A,Y).
move(S1,[X,[C|Y]|S3]):-removeelement(S1,[C|X],S2),
    removeelement(S2,Y,S3).
```

Predikāts *move/3* darbībā:

```
?- move([[b,c,a],[],[]],S).

S = [[c,a],[b],[]] ? a

S = [[c,a],[b],[]]

no
```

Tiek atgriezti 2 vienādi rezultāti, jo sākumā ir 2 tukšas kolonnas, un pārlikšana uz katru no tām ir atsevišķs gadījums.

16.3.4. Gala risinājuma iegūšana

Risinājumu iegūst, tik ilgi pielietojot predikātu *move/2*, līdz tiek iegūts vēlamais rezultāts (*cubesok/1*), kā rezultāts tiek atgriezta vai izdrukāta stāvokļu virkne, kā līdz rezultātam nonākts.

Lai saliktu vairāku soļu izpildi pēc kārtas, jāatrisina vēl viena problēma – tā kā stāvokļu grafs ir ciklisks (ejot pa to, iespējams atgriezties jau apciemotā stāvoklī), pastāv ieciklošanās draudi, staigājot pa stāvokļiem.

Atkarībā no situācijas, ir iespējami šādi varianti:

- izvēlas apstaigāšanas stratēģiju, kas izslēdz ieciklošanos,
- uzskaita soļu skaitu (grafa apstaigāšanas koka dziļumu) un neļauj pārkāpt maksimālajam skaitam,
- uzskaita apstaigātos stāvokļus, neļaujot vēlreiz nonākt tajos.

Izvēlēsimies 3. variantu, jo mums vienalga rezultātā nepieciešams pārstaigāto stāvokļu saraksts:

```
notcontains([],_):-!.
notcontains([B|C],A):-A\=B, notcontains(C,A).
printlist([]).
printlist([A|X]):-write(A),nl,printlist(X).
solve(S):-solve(S,[S]).
solve(S,L):-cubesok(S), !, printlist(L).
solve(S,E):-move(S,T), notcontains(E,T), solve(T,[T|E]).
```

Papildus izmantoti predikāti *notcontains/2*, kas izpildās veiksmīgi, ja saraksts nesatur doto elementu un jau iepriekš izmantotais *printlist/1*.

Izsaukts tiek predikāts *solve/1*, kam padod sākotnējo konfigurāciju, tas savukārt izsauc *solve/2*, kas otrajā pozīcijā padod tukšu sarakstu, kur uzkrās apstaigātos stāvokļus, kuri būs vajadzīgi gan beigu izdrukai, gan, lai izvairītos no atkārtotas apstaigāšanas:

```
?- solve([[b,c,a],[],[]]).
[[],[a,b,c],[]]
[[a],[b,c],[]]
[[],[b,a],[c]]
[[c],[b],[a]]
[[],[b,c],[a]]
[[b],[c],[a]]
[[],[c,b],[a]]
[[c],[a],[b]]
[[],[a,c],[b]]
[[a],[c],[b]]
[[],[b],[c,a]]
[[b],[c,a],[]]
[[c,b],[a],[]]
[[],[a,c,b],[]]
[[a],[c,b],[]]
[[c,a],[b],[]]
[[b,c,a],[],[]]

true ?

(125 ms) yes
```

Rezultāts parāda soļu secību (no apakšas uz augšu), tomēr tas ir ļoti neefektīvs, jo dažādi pēc būtības vienādi stāvokļi tiek uzskatīti par dažādiem, piemēram:

```
[[c,a],[b],[]]
[[b],[c,a],[]]
```

16.3.5. Risinājuma optimizācija

Risinājuma optimizēšanai jāpārraksta predikāts *notcontains/2*, lai atpazītu ekvivalentās konfigurācijas. Vispirms tiek definēts predikāts *notequalstates/2*, kas pasaka, ka divi stāvokļi ir atšķirīgi:

```
equalstates([],[]):-!.
equalstates([A|X],Y):-removeelement(Y,A,Z), !, equalstates(X,Z).
notequalstates(X,Y):-equalstates(X,Y), !, fail; true.
```

Ar piemēriem:

```
?- notequalstates([[a],[b],[c]],[[],[c],[a,b]]).

yes
?- notequalstates([[a,b],[],[c]],[[],[c],[a,b]]).

no
```

Tad, ieliekot, šo predikātu nevienādības vietā, pārdefinējam notcontains/2 par notconyains2/2 un pēc tam *solve/1* un *solve/2* par *solve2/1* un *solve2/2*:

```
notcontains2([],_):-!.
```

```
notcontains2([B|C],A):-notequalstates(A,B), notcontains2(C,A).
solve2(S):-solve2(S,[S]).
solve2(S,L):-cubesok(S), !, printlist(L).
solve2(S,E):-move(S,T), notcontains2(E,T), solve2(T,[T|E]).
```

Un rezultāts ir īsāks:

```
?- solve2([[b,c,a],[],[]]).
[[],[a,b,c],[]]
[[],[b,c],[a]]
[[b],[c],[a]]
[[a],[c,b],[]]
[[c,a],[b],[]]
[[b,c,a],[],[]]

true ?
```

Kādā no nākamajiem risinājuma variantiem nonākam arī pie optimālā risinājuma šim gadījumam – 4 soļos:

```
true ? ;
[[],[a,b,c],[]]
[[],[b,c],[a]]
[[a],[c],[b]]
[[c,a],[b],[]]
[[b,c,a],[],[]]
```

16.4. Nodaļā apskatīto predikātu kopsavilkums

Att. 16-5. Nodaļā apskatīto predikātu kopsavilkums (*examples.pl*).

```
bubblesort([],[]):-!.
bubblesort(X,Z):-shuffle(X,Y,N), N>0, !, bubblesort(Y,Z); Z=X.
shuffle([A],[A],0):-!.
shuffle([A,B|X],[B|Z],N):-A>B, !, Y=[A|X], shuffle(Y,Z,M), N is
    M+1.
shuffle([A|X],[A|Y],N):-shuffle(X,Y,N).
splitlist([],[],[]):-!.
splitlist([A],[A],[]):-!.
splitlist([A,B|X],[A|Y],[B|Z]):-splitlist(X,Y,Z).
mergelist([],[],[]):-!.
mergelist([],A,A):-!.
mergelist(A,[],A):-!.
mergelist([A|X],[B|Y],[A|Z]):-A<B, !, mergelist(X,[B|Y],Z).
mergelist(X,[B|Y],[B|Z]):-mergelist(X,Y,Z).
mergesort([],[]):-!.
mergesort([A],[A]):-!.
mergesort(X,Y):-splitlist(X,X1,X2), mergesort(X1,Y1),
    mergesort(X2,Y2), mergelist(Y1,Y2,Y).
addnode([],B,[[],B,[]]):-!.
addnode([X,B,Y],B,[X,B,Y]):-!.
addnode([X,B,Y],A,[Z,B,Y]):-A<B, !, addnode(X,A,Z).
addnode([X,B,Y],C,[X,B,Z]):-addnode(Y,C,Z).
addnodes([],T,T).
```

```
addnodes([A|X],R,T):-addnode(R,A,S), addnodes(X,S,T).
addnodes(L,T):-addnodes(L,[],T).
printtree([]):-!.
printtree([L,K,R]):-printtree(L), write(K), tab(1),
    printtree(R).
cubesok([[a,b,c],[],[]]):-!.
cubesok([[],[a,b,c],[]]):-!.
cubesok([[],[],[a,b,c]]).
removeelement([A|X],A,X).
removeelement([B|X],A,[B|Y]):-removeelement(X,A,Y).
move(S1,[X,[C|Y]|S3]):-removeelement(S1,[C|X],S2),
    removeelement(S2,Y,S3).
notcontains([],_):-!.
notcontains([B|C],A):-A\=B, notcontains(C,A).
printlist([]).
printlist([A|X]):-write(A),nl,printlist(X).
solve(S):-solve(S,[S]).
solve(S,L):-cubesok(S), !, printlist(L).
solve(S,E):-move(S,T), notcontains(E,T), solve(T,[T|E]).
equalstates([],[]):-!.
equalstates([A|X],Y):-removeelement(Y,A,Z), !, equalstates(X,Z).
notequalstates(X,Y):-equalstates(X,Y), !, fail; true.
notcontains2([],_):-!.
notcontains2([B|C],A):-notequalstates(A,B), notcontains2(C,A).
solve2(S):-solve2(S,[S]).
solve2(S,L):-cubesok(S), !, printlist(L).
solve2(S,E):-move(S,T), notcontains2(E,T), solve2(T,[T|E]).
```

16.5. Vingrinājumi

Vingrinājums #1.

Definēt predikātu *insertsorted/3*, kas sakārtotā sarakstā pareizā vietā ieliek jaunu elementu:

```
?- insertsorted([1,3,4,5],2,X).

X = [1,2,3,4,5]

yes
?- insertsorted([1,3,4,5],0,X).

X = [0,1,3,4,5]

yes
?- insertsorted([1,3,4,5],7,X).

X = [1,3,4,5,7]

yes
```

Vingrinājums #2.

Pēc līdzības ar *printtree/1* (16.2.4) definēt predikātu *treetolist/2*, kas no bināra meklēšanas koka izveido sarakstu ar elementiem augošā secībā:

```
?- addnodes([9,6,12,4,8,11,15,1,10,7],T),treetolist(T,L).

L = [1,4,6,7,8,9,10,11,12,15]
T = [[[[[],1,[]],4,[]],6,[[[],7,[]],8,[]]],9,[[[[],10,[]],
    11,[]],12,[[],15,[]]]]

yes
```

Vingrinājums #3.

Lapas ir tādi mezgli, kam nav bērnu (ne kreisā, ne labā). Pēc līdzības ar *printtree/1* (16.2.4) definēt predikātu *printleaves/1*, kas izdrukā binära koka lapas:

```
?- addnodes([9,6,12,4,8,11,15,1,10,7],T),printleaves(T).
1 7 10 15

T = [[[[[],1,[]],4,[]],6,[[[],7,[]],8,[]]],9,[[[[],10,[]],
    11,[]],12,[[],15,[]]]]

yes
```