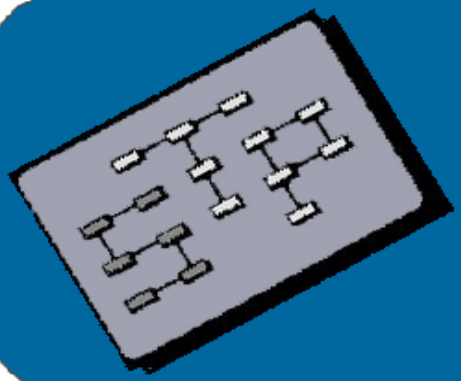
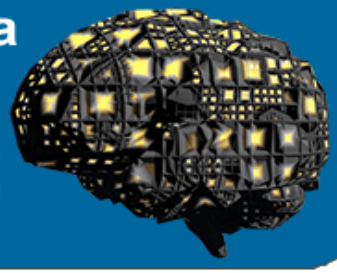


Datorzinātnes un informācijas tehnoloģijas fakultāte

Sistēmu teorijas un projektēšanas katedra

MĀKSLĪGĀ INTELEKTA PAMATI



2. Modulis "Neinformētas pārmeklēšanas stratēģijas stāvokļu telpā"

2.3. Tēma

Stāvokļu telpas neinformēti pārmeklēšanas algoritmi

Dr.habil.sc.ing., profesors **Jānis Grundspeņķis**, Dr.sc.ing., lektore **Alla Anohina**

Sistēmu teorijas un projektēšanas katedra
Datorzinātnes un informācijas tehnoloģijas fakultāte
Rīgas Tehniskā universitāte
E-pasts: *{janis.grundspenkis, alla.anohina}@rtu.lv*
Kontaktadrese: *Meža iela 1/4- {550, 545}, Rīga, Latvija, LV-1048*
Tālrunis: *(+371) 67089{581, 595}*

Tēmas mērķi un uzdevumi

Tēmas mērķis ir sniegt zināšanas par trim stāvokļu telpas pārmeklēšanas algoritmiem un prasmes to realizācijā.

<u>Pēc šīs tēmas apgūšanas Jūs:</u>

- zināsiet pārmeklēšanas atkāpjoties darbību un spēsiet to demonstrēt ar sarakstu palīdzību;
- zināsiet pārmeklēšanas metožu kategorijas un metodes, kas pieder katrai kategorijai;
- zināsiet pārmeklēšanas plašumā un dziļumā darbību, kā arī šo metožu priekšrocības un trūkumus;
- būsiet spējīgi demonstrēt pārmeklēšanas plašumā un dziļumā darbības realizāciju ar sarakstu Open un Closed palīdzību;
- pratīsiet izvērtēt un izvēlēties piemērotu neinformētu pārmeklēšanas stratēģiju konkrētiem praktiskiem uzdevumiem.

Pārmeklēšana atkāpjoties (1)

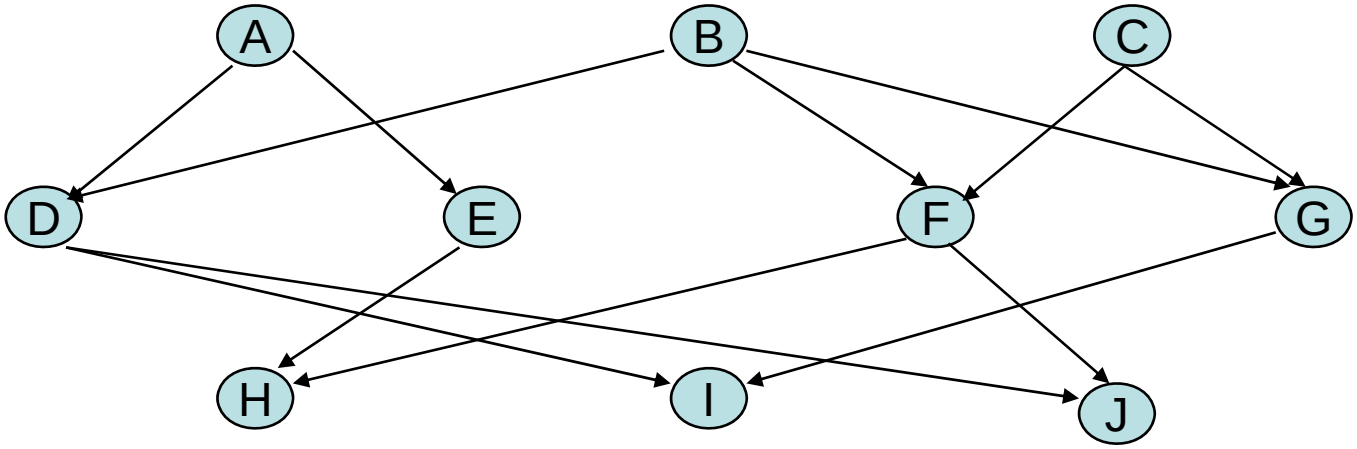
Pārmeklēšana atkāpjoties ir pamattehnika, kas nodrošina virzību uz priekšu stāvokļu telpā un sistemiskumu jeb nonākšanas agrāk apmeklētajās virsotnēs izslēgšanu.

Algoritma būtība:

- Sākas sākumvirsotnē un turpinās tik ilgi, kamēr tekošais ceļš vai nu sasniedz mērķi vai strupceļa virsotni, kas nav mērķis
- Ja tas atrod mērķi, tad darbu beidz un izdod atrisinājuma ceļu
- Ja sasniedz strupceļa virsotni, tad kāpjas atpakaļ uz vistuvāko virsotni tekošajā ceļā, kurai ir neapskatīta pēcteča virsotne, un turpina savu darbību attiecībā pret šo virsotni
- Algoritms turpina savu darbību līdz brīdim, kad atrod mērķi vai kad pārmeklē visu stāvokļu telpu, mērķi neatrodot

Pārmeklēšana atkāpjoties (2)

Piemērs:



Dots: B- sākumstāvoklis H- mērķis

B->D->I (strupceļš un nav mērķis; algoritms atkāpjas uz D)

B->D->J (strupceļš un nav mērķis; algoritms atkāpjas uz D)

B->D (D virsotnei vairs nav pēcteču; algoritms atkāpjas uz B)

B->F->H (mērķis ir sasniegts; algoritms darbu beidz)

Pārmeklēšana atkāpjoties (3)

Šī algoritma realizācijai izmanto 3 sarakstus un vienu mainīgo:

SL (State List) - stāvokļu saraksts, kurā ieraksta tekošā ceļa stāvokļus; kad atrod mērķi, tad šis saraksts satur sakārtotu virsotņu sarakstu, kas veido atrisinājuma ceļu

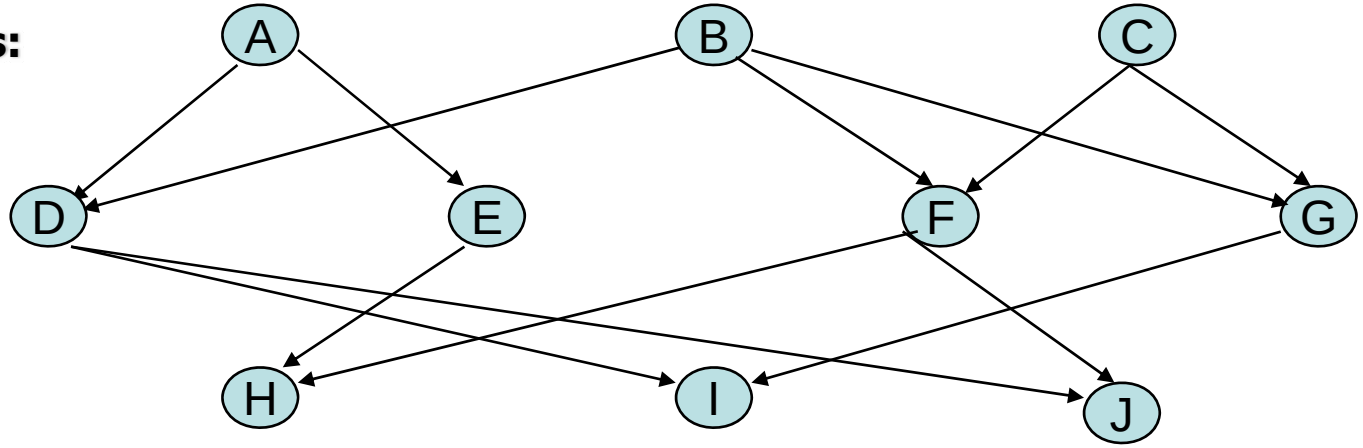
NSL (New State List) - jaunu stāvokļu saraksts, kurš satur virsotnes, kas gaida izvēršanu

DE (Dead Ends) - strupceļu virsotņu saraksts, kurā ieraksta stāvokļus, kuru pēcteči nesatur mērķi; ja pārmeklēšanas gaitā atkal parādās stāvoklis, kas jau ir šajā sarakstā, tad atkārtoti to neapskata

CS (Current State) - mainīgais, kas satur tekošo virsotni, kuras pēcteči tiek ģenerēti

Pārmeklēšana atkāpjoties (4)

Piemērs:



Dots: B- sākumstāvoklis H- mērķis

Iterācija	NSL	CS	SL	DE
0	B	B	B	∅
1	D F G B	D	D B	∅
2	I J D F G B	I	I D B	∅
3	J D F G B	J	J D B	I
4	D F G B	D	D B	J I
5	F G B	F	F B	D J I
6	H F G B	**H**	H F B	D J I

Mērķis ir sasniegts

Pārmeklēšana atkāpjoties (5)

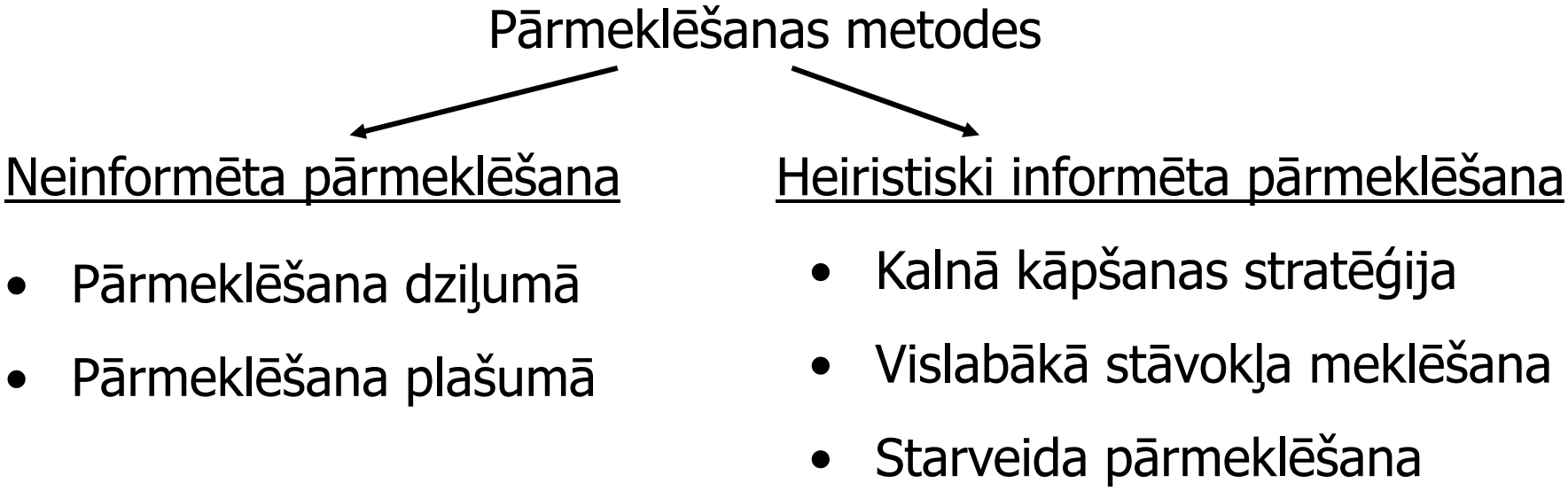
```
function backtracking;
begin
//sarakstu inicializācija
SL:= [problēmas_risināšanas_sākuma_stāvoklis];
NSL:= [problēmas_risināšanas_sākuma_stāvoklis];
DE:= [];
CS:= problēmas_risināšanas_sākuma_stāvoklis;
//kamēr ir virsotnes, kuras var tikt apskatītas, realizē ciklu
while (NSL ≠ [])  do
  begin
    //ja tekoša virsotne ir mērķa stāvoklis, tad funkcija beidz darbību un atgriež atrisinājuma ceļu, kas glabājas sarakstā SL
     if CS = mērķa_stāvoklis then return (SL);
     if CS nav pēcteču, izņemot tos, kas jau ir sarakstos DE, SL,vai NSL then
       begin
         //ciklā realizē atkāpšanos
          while (SL ≠ []) and (CS = pirmais elements no SL) do
             begin
                  pievieno CS sarakstam DE;
                  izslēdz pirmo elementu no SL;
                  izslēdz pirmo elementu no NSL;
                  CS:= pirmais elements no NSL;
             end;
           pievieno CS sarakstam SL;
       end
     else
       begin
          pievieno CS pēctečus (izņemot tos, kas jau ir sarakstos DE, SL, vai NSL) sarakstam NSL;
          CS:= pirmais elements no NSL;
          pievieno CS sarakstam SL;
       end;
   end;
//ja visa stāvokļu telpa tika pārmeklēta un mērķa stāvoklis netika atrasts, atgriež neveiksmi
return (neveiksme);
end;
```

Pārmeklēšana atkāpjoties (6)

Citi pārmeklēšanas algoritmi, kas tiek apskatīti tālāk šajā mācību kursā, izmanto pārmeklēšanas atkāpjoties idejas, tādas kā:

- Tiek veidots neizvērsto stāvokļu saraksts (NSL), lai būtu iespēja atgriezties pie jebkura no tiem pārmeklēšanas gaitā
- Tiek uzturēts sturpceļa stāvokļu saraksts (DE), lai novērstu nelietderīgu ceļu pārbaudi
- Tiek uzturēts tekošā ceļa saraksts (SL), kas tiek atgriezts sasniedzot mērķi
- Pārbauda katra jauna stāvokļa esamību šajos sarakstos, lai novērstu ciklus algoritma darbībā

Pārmeklēšanas metožu klasifikācija



Neinformētajā jeb ***aklajā pārmeklēšanā*** nav nekādas citas papildinformācijas par stāvokļu labumu (kvalitāti), kā tikai pati stāvokļu telpa. Tātad, nav iespējams pateikt, ka viens stāvoklis ir labāks par citu. Šajā gadījumā pārmeklēšanas efektivitāti nosaka tikai stāvokļu telpas raksturs.

Pārmeklēšana plašumā un dziļumā

Pārmeklēšanas stratēģijas pēc stāvokļu apmeklēšanas kārtības dalās:

- Pārmeklēšanā plašumā
- Pārmeklēšanā dziļumā

Meklējot dziļumā algoritms nolaižas pēc iespējas dziļāk stāvokļu telpā, cenšoties atrast mērķi. Ja tas neizdodas, algoritms atkāpjas uz tuvāko virsotni tekošajā ceļā, kuras pēctecis vēl nav apskatīts, un atkārto savu darbību.

Meklējot plašumā algoritms tieši pretēji pēta stāvokļu telpu līmeni pa līmenim. Apskatot kādu virsotni, tas tūlīt ģenerē visus tās pēctečus un starp tiem meklē mērķi. Ja mērķi neatrod, tad ģenerē pirmā pēcteča pēctečus un pārbauda tos. Ja starp tiem mērķis nav atrasts, ģenerē otrā pēcteča pēctečus u.t.t.

Pārmeklēšana plašumā

Priekšrocības:

+ Garantē atrisinājuma atrašanu, jo realizē izsmeļošu stāvokļu telpas pārmeklēšanu
+ Nezaudē atrisinājumus, kas atrodas augstākos līmeņos
+ Vienmēr garantē, ka atrisinājums tiks sasniegts pa iespējami īsāko ceļu, jo pirms pārbauda stāvokļus i+1-ajā līmenī tiek apskatīti visi stāvokļi i-tajā līmenī
+ Algoritms vislabāk ir piemērots stāvokļu telpām, kurās zarošanās koeficients ir neliels

Trūkumi:

– Algoritmam nav nekādas informācijas, kas tam palīdzētu virzīties uz atrisinājumu stāvokļu telpā
– Algoritms ir nepiemērots stāvokļu telpām ar lielu zarošanās koeficientu, jo tad ir jāapmeklē liels skaits virsotņu
– Algoritms ir neefektīvs, kad atrisinājums atrodas dziļos līmeņos

Pārmeklēšana dziļumā

Priekšrocības:

+ Garantē atrisinājuma atrašanu, jo realizē izsmeļošu stāvokļu telpas pārmeklēšanu
+ Algoritms ir efektīvs, kad atrisinājums atrodas dziļos līmeņos

Trūkumi:

– Algoritmam nav nekādas informācijas, kas tam palīdzētu virzīties uz atrisinājumu stāvokļu telpā
– Algoritms ir nepiemērots lielām stāvokļu telpām, jo var neiegūt atrisinājumu pieļaujamā laikā
– Algoritms ir neefektīvs, kad atrisinājums atrodas augstākos līmeņos
– Algoritms negarantē mērķa atrašanu pa īsāko ceļu

Pārmeklēšanas plašumā un dziļumā realizācija (1)

Pārmeklēšanu dziļumā un plašumā realizē, lietojot divus sarakstus:

- OPEN
  - Līdzīgs sarakstam NSL, kas tiek izmantots pārmeklēšanā atkāpjoties
  - Šajā sarakstā ieraksta stāvokļus, kas ir ģenerēti, bet kuru pēcteči nav apskatīti
  - Kārtība, kurā stāvokļus ievieto šajā sarakstā, nosaka pārmeklēšanas kārtību

- CLOSED
  - Apvieno sevī sarakstus DE un SL, kas tiek izmantoti pārmeklēšanā atkāpjoties
  - Šajā sarakstā ieraksta stāvokļus, kas jau ir apskatīti

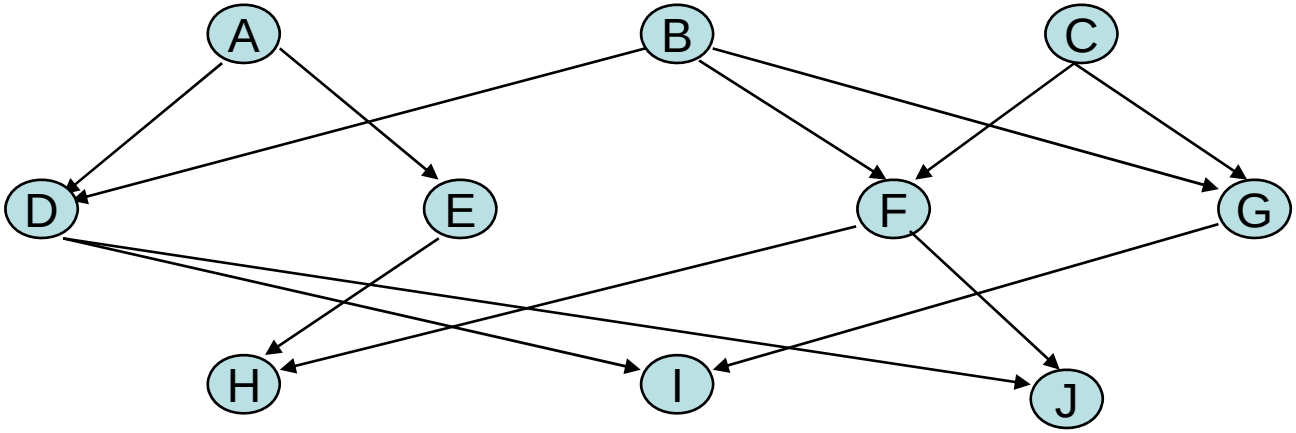
Pārmeklēšanas plašumā un dziļumā realizācija (2)

Realizējot pārmeklēšanu dziļumā un plašumā, lietojot sarakstus OPEN un CLOSED, ir jāņem vērā:

- Stāvokļus izslēdz no saraksta OPEN kreisās puses
- Jaunus stāvokļus ievieto saraksta OPEN labajā pusē, meklējot plašumā, bet kreisajā pusē, meklējot dziļumā
- Algoritms darbu beidz, vai nu tad, kad mērķa stāvoklis atrodas saraksta OPEN pirmajā pozīcijā, vai kad tas ir ievietots sarakstā CLOSED
- Stāvokļus, kuri jau atrodas OPEN vai CLOSED sarakstā, vairs nepievieno sarakstam OPEN

Pārmeklēšanas plašumā un dziļumā realizācija (3)

Piemērs: No datiem virzīta pārmeklēšana plašumā



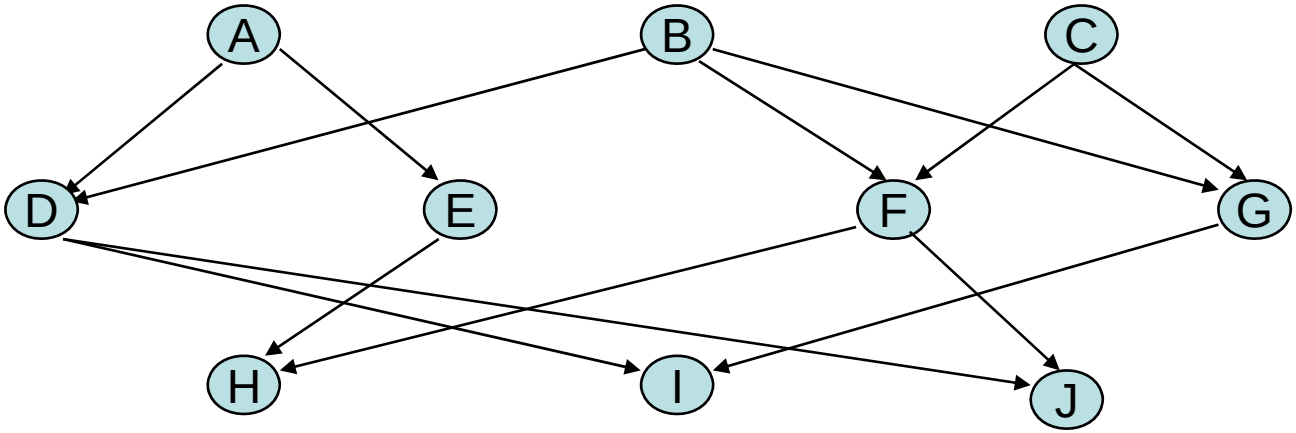
Dots: B- sākumstāvoklis H- mērķis

Iterācija	OPEN	CLOSED
0	B	∅
1	D F G	B
2	F G I J	B D
3	G I J H	B D F
4	I J H	B D F G
5	J H	B D F G I
6	**H**	B D F G I J

Mērķis ir sasniegts

Pārmeklēšanas plašumā un dziļumā realizācija (4)

Piemērs: No mērķa virzīta pārmeklēšana plašumā



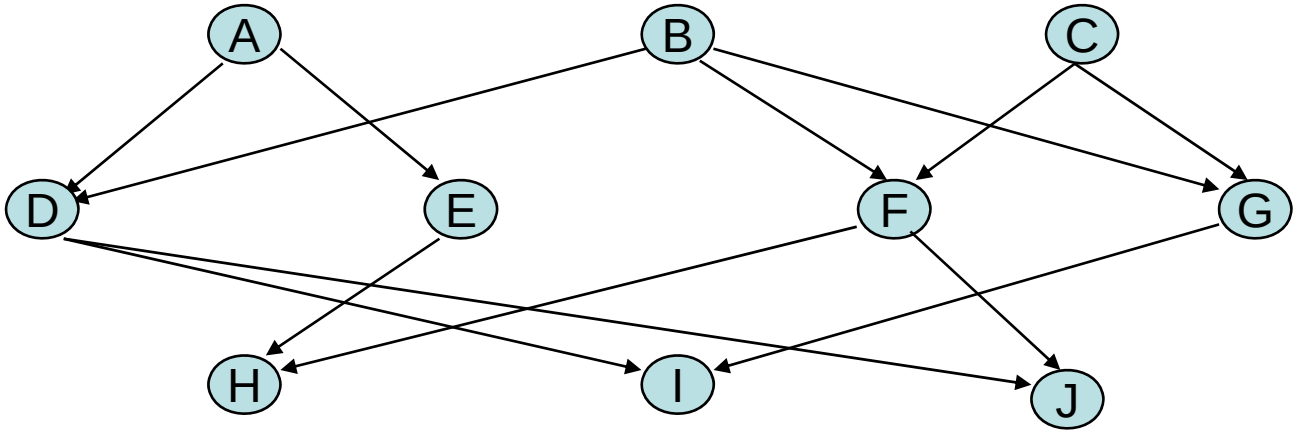
Dots: H- sākumstāvoklis B- mērķis

Iterācija	OPEN	CLOSED
0	H	∅
1	E F	H
2	F A	H E
3	A B C	H E F
4	**B** C	H E F A

Mērķis ir sasniegts

Pārmeklēšanas plašumā un dziļumā realizācija (5)

Piemērs: No datiem virzīta pārmeklēšana dziļumā



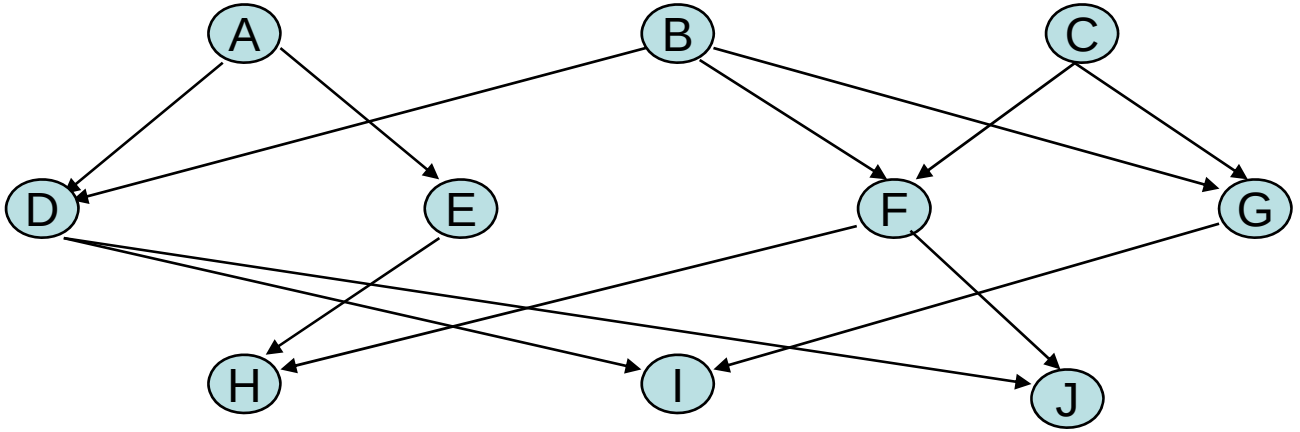
Dots: B- sākumstāvoklis H- mērķis

Iterācija	OPEN	CLOSED
0	B	∅
1	D F G	B
2	I J F G	B D
3	J F G	B D I
4	F G	B D I J
5	**H** G	B D I J F

Mērķis ir sasniegts

Pārmeklēšanas plašumā un dziļumā realizācija (6)

Piemērs: No mērķa virzīta pārmeklēšana dziļumā



Dots: H- sākumstāvoklis B- mērķis

Iterācija	OPEN	CLOSED
0	H	∅
1	E F	H
2	A F	H E
3	F	H E A
4	**B** C	H E A F

Mērķis ir sasniegts

Pārmeklēšanas plašumā un dziļumā realizācija (7)

```
function pārmeklēšana_plašumā;
begin
OPEN := [problēmas_risināšanas_sākuma_stāvoklis];
CLOSED := [];
while (OPEN ≠ []) do
    begin
        izslēdz pirmo elementu X no OPEN saraksta;
        if X ir mērķa_stāvoklis then return (atrasts);
        ģenerē visus X pēctečus;
        ievieto X sarakstā CLOSED;
        izslēdz jebkuru X pēcteci, ja tas jau ir sarakstā OPEN vai CLOSED, lai novērstu ciklus;
        atlikušos X pēctečus ievieto to atrašanas kārtībā saraksta OPEN labajā pusē;
    end;
//ja visa stāvokļu telpa tika pārmeklēta un mērķa stāvoklis netika atrasts, atgriež neveiksmi
return (neveiksme);
end;
```

Pārmeklēšanas plašumā un dziļumā realizācija (8)

```
function pārmeklēšana_dziļumā;
begin
OPEN := [problēmas_risināšanas_sākuma_stāvoklis];
CLOSED := [];
while (OPEN ≠ []) do
    begin
        izslēdz pirmo elementu X no OPEN saraksta;
        if X ir mērķa_stāvoklis then return (atrasts);
        ģenerē visus X pēctečus;
        ievieto X sarakstā CLOSED;
        izslēdz jebkuru X pēcteci, ja tas jau ir sarakstā OPEN vai CLOSED, lai novērstu ciklus;
        atlikušos X pēctečus ievieto to atrašanas kārtībā saraksta OPEN kreisajā pusē;
    end;
//ja visa stāvokļu telpa tika pārmeklēta un mērķa stāvoklis netika atrasts, atgriež neveiksmi
return (neveiksme);
end;
```

Pārmeklēšanas plašumā un dziļumā realizācija (9)

Realizējot pārmeklēšanu plašumā, saraksts OPEN tiek veidots kā rinda (darbojas pēc principa FIFO- First In - First Out), pārmeklēšanā dziļumā tas tiek veidots kā steks (darbojas pēc principa LIFO- Last In - First Out).

Ne saraksts OPEN, ne CLOSED neglabā atrisinājuma ceļu. Ja atrisinājuma ceļš ir vajadzīgs, tad sarakstos ir jāievieto nevis tekošie stāvokļi, bet stāvokļu pāri:

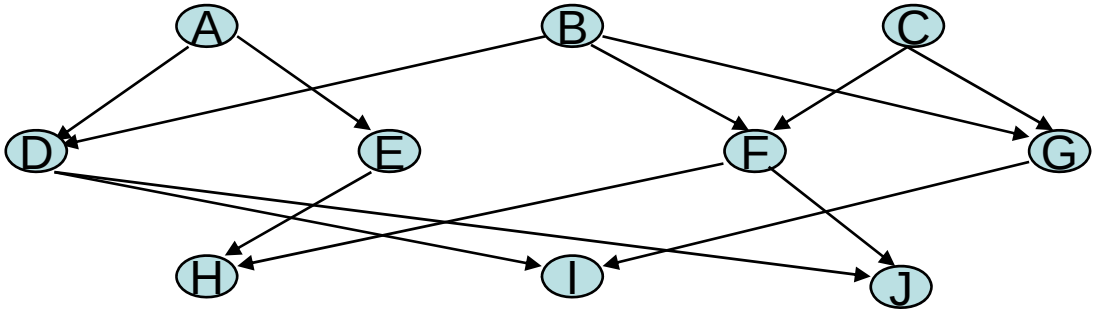
[stāvoklis, priekštecis] – realizējot no datiem virzītu pārmeklēšanu

[stāvoklis, pēctecis] – realizējot no mērķa virzītu pārmeklēšanu

Turpretim, pārmeklēšanas atkāpjoties algoritmā atrisinājuma ceļu glabā saraksts SL.

Pārmeklēšanas plašumā un dziļumā realizācija (10)

Piemērs: No datiem virzīta pārmeklēšana plašumā, saglabājot atrisinājuma ceļu



Dots: B- sākumstāvoklis H- mērķis

Iterācija	OPEN	CLOSED
0	[B,∅]	∅
1	[D, B] [F, B] [G, B]	[B,∅]
2	[F, B] [G, B] [I, D] [J, D]	[B,∅] [D, B]
3	[G, B] [I, D] [J, D] [H, F]	[B,∅] [D, B] [F, B]
4	[I, D] [J, D] [H, F]	[B,∅] [D, B] [F, B] [G, B]
5	[J, D] [H, F]	[B,∅] [D, B] [F, B] [G, B] [I, D]
6	[**H**, F]	[B,∅] [D, B] [F, B] [G, B] [I, D] [J, D]

Mērķis ir sasniegts

Atrisinājuma ceļš: [H, F] [F, B] [B,∅] - B->F->H