University of Technology

الجامعة التكنولوجية

Computer Science Department

قسم علوم الحاسوب

Searching Strategies

استراتيجيات البحث

Assist Prof. Dr. Mustafa Jaim Hadi

ا.م.د. مصطفى جاسم هادي

Second Stage- First Course

cs.uotechnology.edu.iq

Database and Compound Objects

Database:

Example1:

1- Assert predicate:

- assert(X) or assertz(X) :Adds a new fact to the database. Term is asserted as the last fact with the same key predicate.
  - ✓ For example;

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
    assert(person ("Ali")),
    assert(person ("Zaki")),
    assert(person ("Suha")),
    findall(X,person(X),L).

goal: list_preson(L).
%L=["Ali","Zaki","Suha"]
```

- asserta(X) :Same as assert, but adds a fact X at the beginning of the database.
  - ✓ For example;

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
        asserta(person ("Ali")),
        asserta(person ("Zaki")),
        asserta(person ("Suha")),
        findall(X,person(X),L).
```

goal: list_preson(L). %L=["Suha","Zaki","Ali"]

2- **Retract predicate:**

- retract(X): Removes a fact X from the database.
  - ✓ For example;

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
        assert(person ("Ali")),
        assert(person ("Zaki")),
        assert(person ("Suha")),
        retract(person ("Zaki")),
        findall(X,person(X),L).

goal: list_preson(L).
%L=["Ali","Suha"]
```

- retractall(X): Removes all facts from the database for which the head unifies with X.
  - ✓ For example;

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
clauses
list_preson(L):-
        assert(person ("Ali")),
        assert(person ("Zaki")),
        assert(person ("Suha")),
        retractall(person (_)),% retractall(_),
        findall(X,person(X),L).

goal: list_preson(L).
%L=[]
```

Example2: Insert the following facts to a database then put them in a list. The facts are:
f(1).
f(2).
f(3).

Solution:

```
domains
 i=integer
 f=f(i).
 lf=f*.
database
 f(i).
predicates
 run(lf).
 g(f).
 test
clauses
 test:-
 assertz(f(1)),
 assertz(f(2)),
 assertz(f(3)).

g(f(X)):-f(X).
run(L):-test,
       findall(S,g(S),L).

goal:
   run(X).  %X=[f(1),f(2),f(3)]
```

Example3: Use a database concept to perform the following goal:
Goal: run("He bought 7 oranges their total weight 1.5 kg").
And give the following output:

String=   He   length=   2
String=   bought   length=   6
String=   oranges   length=   7
String=   their   length=   5
String=   total   length=   5
String=   weight   length=   6
String=   kg   length=   2

Solution:

```
database
db_string(String,integer)
predicates
split_tokens(string)
run(string)
print_string
clauses
run(S):-retractall(_),
  split_tokens(S),
  print_string.

split_tokens(S):-
   fronttoken(S,W,R),
   isname(W),!,str_len(W,N),
   assert(db_string(W,N)),
   split_tokens(R).
split_tokens(S):-
   fronttoken(S,_,R),!,split_tokens(R).
split_tokens("").

print_string:-
db_string(S,N),write("String=   ",S,"      length=   ",N),nl,fail.
print_string.

goal
    run("He bought 7 oranges  their total  weight 1.5 kg").
/*String=   He      length=   2
String=   bought      length=   6
String=   oranges      length=   7
String=   their      length=   5
String=   total      length=   5
String=   weight      length=   6
String=   kg      length=   2 yes         */
```

Compound Objects

Example1:

```
domains
   predecessor=parent(father,son);child(string).
   father=father(string).
   son=son(predecessor).
predicates
  father(string,string).
  grandfather(predecessor).
clauses
  father("Ali","Zaki").
  father("Zaki","Suha").

  grandfather(parent(X,son(parent(Y,son(Z))))):-
    father(X1,Y1),
    father(Y1,Z1),
    X=father(X1),
    Y=father(Y1),
    Z=child(Z1).

goal:
    grandfather(X).
    %X=parent(father("Ali"),son(parent(father("Zaki"),son(child("Suha")))))
```

Example2:

```
domains
st=s(symbol,integer)
l=st*
predicates
member(st,l)
clauses
member(s(A,_),[s(A,_)|_]):-!.
member(X,[_|T]):-
        member(X,T).

goal
   member(s(a,5),[s(c,5),s(a,3),s(d,6)]).%yes
```

Example3:

```
domains
st=s(symbol,integer)
l=st*
predicates
member(st,l,integer,integer)
clauses

member(s(A,_),[s(A,_)|_],N,N):-!.
member(X,[_|T],N1,N):-
        N2=N1+1,member(X,T,N2,N).

goal
% member(s(a,5),[s(c,5),s(a,3),s(d,6)],1,N).%N=2
member(s(a,5),[s(c,5),s(g,3),s(d,6),s(a,7)],1,N).%N=4
%member(s(a,2),[s(a,5),s(g,3),s(d,6),s(c,7)],1,N).%N=1
%member(s(f,2),[s(a,5),s(g,3),s(d,6),s(c,7)],1,N).%No Solution
```

Example4:

```
domains
st=s(symbol,integer)
l=st*
predicates
member(st,l,st)
clauses
member(s(A,_),[s(A,X)|_],s(A,X)):- !.
member(X,[_|T],Z):-
        member(X,T,Z).

goal
   member(s(a,5),[s(c,5),s(a,3),s(d,6)],X).%X=s("a",3)
   %member(s(a,3),[s(c,5),s(a,5),s(d,6)],X).%X=s("a",5)
```

```
%member(s(d,1),[s(c,5),s(a,5),s(d,6)],X).%X=s("d",6)
%member(s(c,1),[s(c,5),s(a,5),s(d,6)],X).%X=s("c",5)
```

Example5:

```
domains
st=s(symbol,integer)
l=st*
i=integer
predicates
del(st,l,l)
clauses
del(s(A,_),[s(A,_)|L],L):-!.
del(X,[H|T],[H|Z]):-
  del(X,T,Z).
goal:
%del(s(g,9),[s(a,5),s(g,3),s(d,6),s(c,7)],X).%X=[s("a",5),s("d",6),s("c",7)]
```

Example7:

```
domains
st=s(symbol,integer)
l=st*
predicates
difference(l,l,l)
member(st,l)
clauses
difference([],_,[]):-!.
difference([H|T],Z,[H|T1]):-
              not(member(H,Z)),!,
              difference(T,Z,T1).

difference([_|T],X,Y):-
  difference(T,X,Y).

member(s(A,_),[s(A,_)|_]):-!.
member(X,[_|T]):-
     member(X,T).

goal
 difference([s(k,3),s(a,0),s(f,2),s(b,6)],[s(d,10),s(b,8),s(a,5),s(g,9)],X).
 % X=[s("k",3),s("f",2)]
```

Forward Chaining

/* A prolog program that applies the concept of forward chaining system in animal classification. */

```
domains
    s=symbol
database
    have_found(s)
    db_confirm(s,s)
    db_denied(s,s)
predicates
    guess_animal
    find_animal
    test1(s)
    test2(s,s)
    test3(s,s,s)
    test4(s,s,s,s)
    it_is(s)
    confirm(s,s)
    remember(s,s,s)
    check_if(s,s)
clauses
guess_animal:-
    find_animal,
    have_found(X),write("Your animal is a(n)",X),!.

find_animal:-
    test1(X),test2(X,Y),test3(X,Y,Z),test4(X,Y,Z,_),!.
find_animal.

test1(m):-
    it_is(mammal),!.
test1(n).

test2(m,c):-
    it_is(carnivorous),!.
test2(m,n).
test2(n,w):-
    confirm(does,swim),!.
test2(n,n).

test3(m,c,s):-
     confirm(has,stripes),
     asserta(have_found(tiger)),!.
test3(m,c,n):-
     asserta(have_found(cheetah)).
```

```
test3(m,c,l):-
        not(confirm(does,swim)),not(confirm(does,fly)),!.
test3(m,n,n):-
        asserta(have_found(blue_whale)).
test3(n,n,f):-
        confirm(does,fly),asserta(have_found(eagle)),!.
%test3(n,n,n):-
        %asserta(have_found(ostrich)).
test3(n,w,t):-
        confirm(has,tentacles),asserta(have_found(octopus)),!.
test3(n,w,n).

test4(m,n,l,s):-
        confirm(has,stripes),asserta(have_found(zebra)),!.
test4(m,n,l,n):-
        assert(have_found(giraffe)),!.
test4(n,w,n,f):-
        confirm(has,feather),asserta(have_found(penguin)),!.
test4(n,w,n,n):-
        asserta(have_found(sardine)),!.
test4(n,n,n,n):-
        retractall(_),write("Sorry,your animal is unknown\n").

it_is(mammal):-%لبون
      confirm(has,hair),!.%شَعْر
it_is(mammal):-%لبون
      confirm(does,give_milk).%يعطي الحَلِيب
it_is(ungulate):-%ذو حوافر
      it_is(mammal),%لبون
      confirm(has,hooves),%حوافر
      confirm(does,chew_cud),!.%مجتر
it_is(carnivorous):-
      confirm(has,pointed_teeth),!.%اسنان حادة
it_is(carnivorous):-%آكل اللحوم
      confirm(does,eat_meat),!.%ياكل اللحم
it_is(bird):-%طَائِر
      confirm(has,feathers),%رِيش
      confirm(does,lay_egges),!.%يبيض

confirm(X,Y):-db_confirm(X,Y),!.
confirm(X,Y):-not(db_denied(X,Y)),!,check_if(X,Y).

check_if(X,Y):-write(X),write(" it "),write(Y),nl,
            readln(Reply),remember(X,Y,Reply).

remember(X,Y,yes):-asserta(db_confirm(X,Y)).
```

```
remember(X,Y,no):-asserta(db_denied(X,Y)),fail.

goal:
   guess_animal.
/*has it hair
yes
has it pointed_teeth
yes
has it stripes
yes
Your animal is a(n)tiger
---
has it hair
no
does it give_milk
yes
has it pointed_teeth
no
does it eat_meat
no
Your animal is a(n)blue_whale
-----
has it hair
yes
has it pointed_teeth
no
does it eat_meat
no
Your animal is a(n)blue_whale yes
----
has it hair
no
does it give_milk
no
does it swim
no
does it fly
yes
Your animal is a(n)eagle yes
*/
```

Backward Chaining

/* A prolog program that applies the concept of Backward chaining system in animal classification. */

```
domains
    i=integer
    s=symbol.
    f1=db_confirm(s,s).
    list_f1=f1*.
    f2=db_denied(s,s).
    list_f2=f2*.
database
    db_confirm(s,s)
    db_denied(s,s)
predicates
    identify(s)
    it_is(s)
    confirm(s,s)
    remember(s,s,s)
    check_if(s,s)
    guess_animal.
    get_confirm(f1)
    get_denied(f2)
    n_confirm.
    n_denied.
    length(list_f1,i).
    length(list_f2,i).
    print_list(list_f1).
    print_list(list_f2)

clauses
identify(giraffe):-% زَرافة
    it_is(ungulate),% ذو حوافر
    confirm(has,long_neck),% عنق طويل
    confirm(has,long_legs),% سيقان طويلة
    confirm(has,dark_spots),!.% بقع داكنة
identify(zebra):-% حمار وحشي
    it_is(ungulate),% ذو حوافر
    confirm(has,black_stripes),!.% خطوط سوداء
identify(cheetah):-% فهد
    it_is(mammal),% لبون
    it_is(carnivorous),% آكل اللحوم
    confirm(has,tawny_color),% لون أسمر مصفر
    confirm(has,black_spots),!.% بقع داكنة
identify(triger):-
```

```
    it_is(mammal),%لبون
    it_is(carnivorous),%آكل اللحوم
    confirm(has,tawny_color),%لون أسمر مصفر
    confirm(has,black_strips),!.%بقع داكنة
identify(eagle):-%نَسْر
    it_is(bird),%طَائِر
    confirm(does,fly),%يطير , يحلقَ
    it_is(carnivorous),%آكل اللحوم
    confirm(has,use_as_national_symbol),!.%يستعمل احيانا كرمز وطني

it_is(mammal):-%لبون
    confirm(has,hair),!.%شَعْر
it_is(mammal):-%لبون
    confirm(does,give_milk).%يعطي الحَلِيب
it_is(ungulate):-%ذو حوافر
    it_is(mammal),%لبون
    confirm(has,hooves),%حوافر
    confirm(does,chew_cud),!.%مجتر
it_is(carnivorous):-
    confirm(has,pointed_teeth),!.%اسنان حادة
it_is(carnivorous):-%آكل اللحوم
    confirm(does,eat_meat),!.%ياكل اللحم
it_is(bird):-%طَائِر
    confirm(has,feathers),%رِيش
    confirm(does,lay_egges),!.%يبيض

confirm(X,Y):-
    db_confirm(X,Y),!.
confirm(X,Y):-
    not(db_denied(X,Y)),!,check_if(X,Y).

check_if(X,Y):-
    write(X),write(" it "),write(Y),nl,
    readln(Reply),remember(X,Y,Reply).

remember(X,Y,yes):-assert(db_confirm(X,Y)).
remember(X,Y,no):-assert(db_denied(X,Y)),fail.

guess_animal:-
     identify(X),write("Your animal is a(n)",X),nl,nl,
     n_confirm,
     n_denied,!.
guess_animal:-
     write("Sorry,your animal is unknown"),nl.
```

```
get_confirm(db_confirm(X,Y)):-db_confirm(X,Y).
get_denied(db_denied(X,Y)):-db_denied(X,Y).

n_confirm:-
       findall(S,get_confirm(S),L),
       write("DataBase for correct replies are:\n"),
       print_list(L),
       length(L,X),
       write("Number of correct replies=",X),nl.

n_denied:-
       findall(S,get_denied(S),L),
       write("DataBase for uncorrect replies are:\n"),
       print_list(L),
       length(L,X),
       write("Number of uncorrect replies=",X),nl.

length([],0):-!.
length([_|T],X):-
  length(T,X1),
  X=X1+1.

print_list([]):-!.
print_list([H|T]):-
     write(H),nl,
     print_list(T).

goal:guess_animal.

/*
has it hair
no
does it give_milk
no
has it feathers
yes
does it lay_egges
yes
does it fly
yes
has it pointed_teeth
no
does it eat_meat
yes
has it use_as_national_symbol
yes
```

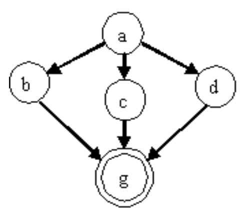
Your animal is a(n)eagle

DataBase for correct replies are:
db_confirm("has","feathers")
db_confirm("does","lay_egges")
db_confirm("does","fly")
db_confirm("does","eat_meat")
db_confirm("has","use_as_national_symbol")
Number of correct replies=5
DataBase for uncorrect replies are:
db_denied("has","hair")
db_denied("does","give_milk")
db_denied("has","pointed_teeth")
Number of uncorrect replies=3
yes
*/
%--

Path Building Using Forward Chaining

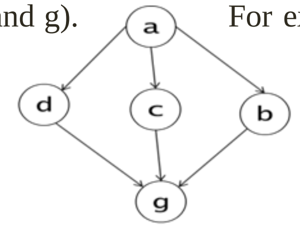
/* A prolog program that applies the concept of Path Building Using Forward Chaining. */

```
predicates
   run(char,char).
   find_rout (char,char).
   path(char,char).
   write_rout.
database
   rout(char,char).
clauses
   run(_,_):-retractall(_),fail.
   run(S,E):-find_rout(S,E),fail.
   run(_,_):-write_rout.

   find_rout(S,E):-path(S,E),asserta(rout(S,E)),!.
   find_rout(S,E):-path(S,M), % Here the cut(!) should be active in case If you want
only one solution.
           find_rout(M,E), asserta(rout(S,M)).

   write_rout:-rout(S,E),
         write("\nSearching from     ", S,"   to   ",E),
         nl,fail.
   write_rout.

   path('a','b').
   path('a','c').
   path('a','d').
   path('b','g').
   path('c','g').
   path('d','g').
```



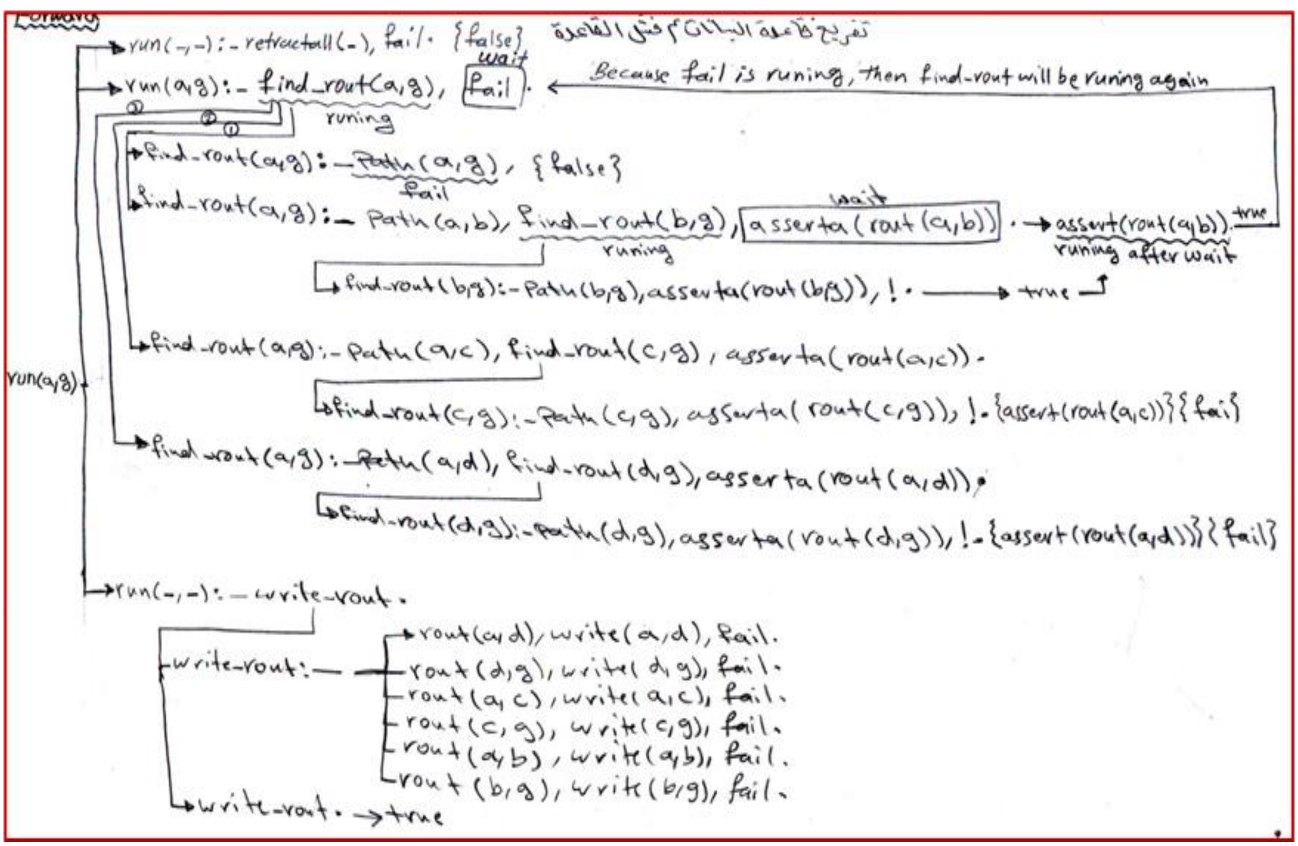

```
/*goal:
           run('a','g').
   Searching from    a  to  d
   Searching from    d  to  g
   Searching from    a  to  c
   Searching from    c  to  g
   Searching from    a  to  b
   Searching from    b  to  g
   yes*/
```

Exercise: There are three routes in the shape below. Write a prolog program (Using a database concept) to move from (a to d and g), (a to c and g), and (a to b and g). For example, take this goal: run('a','g'). The output will be as follows:



Moving from	a	to	d
Moving from	d	to	g
Moving from	a	to	c
Moving from	c	to	g
Moving from	a	to	b
Moving from	b	to	g

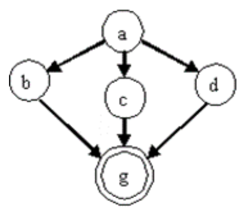
You can benefit from the manual tracing for the above program as in the figure below:



Path Building Using Backward Chaining

/* A prolog program that applies the concept of Path Building Using Backward Chaining. */

```
predicates
   run(char,char).
   find_rout (char,char).
   path(char,char).
   write_rout.
database
   rout (char,char).
clauses
   run(_,_):- retractall(_),fail.
   run(S,E):-find_rout(S,E),fail.
   run(_,_):- write_rout.

   find_rout(S,E):-path(S,E),asserta(rout(S,E)).
   find_rout(S,E):-path(M,E), %Here the cut(!) should be active in case If you want
only one solution.
        find_rout(S,M), asserta(rout(M,E)).

   write_rout:-rout(S,E),
        write(" \nSearching from     ", E,"   t o   ",S),
        nl,fail.
   write_rout.
```



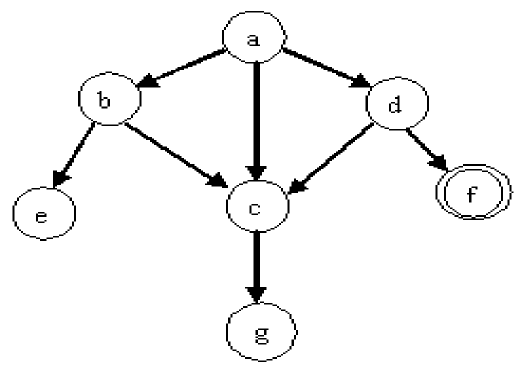

```
    path('a','b').
    path('a','c').
    path('a','d').
    path('b','g').
    path('c','g').
    path('d','g').

/*goal:
          run('a','g').
    Searching from    g  t o  d
    Searching from    d  t o  a
    Searching from    g  t o  c
    Searching from    c  t o  a
    Searching from    g  t o  b
    Searching from    b  t o  a
   Yes */
```

Breadth First Search

/* A prolog program that applies the concept of breadth first search. */

```
domains
    c=char.
    l=c*.
predicates
   breadth(l,l,c).
   difference(l,l,l).
   append(l,l,l).
   member(c,l).
   print(l,l).
   path(c,c).
clauses
   breadth([],_,_):-!,write("Goal is not found ").
   breadth([G|T_Open],Closed,G):-
         !,print([G|T_Open],Closed),write("Goal is found "),nl.
   breadth([H|T_Open],Closed,G):-
        print([H|T_Open],Closed),
        findall(X,path(H,X),Children),
        append(Closed,[H],Closed1),
        difference(Children,T_Open,Children1),
        difference(Children1,Closed1,Children2),
        append(T_Open,Children2,Open1),%Put remaining children on rigth of Open.
        breadth(Open1,Closed1,G).

   difference([],_,[]):- !.
   difference([H|T],Z,[H|T1]):-
        not(member(H,Z)),!,
        difference(T,Z,T1).
   difference([_|T],Z,T1):-
        difference(T,Z,T1).

   member(H,[H|_]):-!.
   member(H,[_|T]):-
        member(H,T).
```

```
append([],L,L):-!.
append([H|T],L,[H|M]):-
     append(T,L,M).

print(Open,Closed):-
     write("Open=",Open,"       ","Closed=",Closed),nl.

path('a','b').
path('a','c').
path('a','d').
path('b','e').
path('b','c').
path('d','c').
path('d','f').
path('c','g').
```



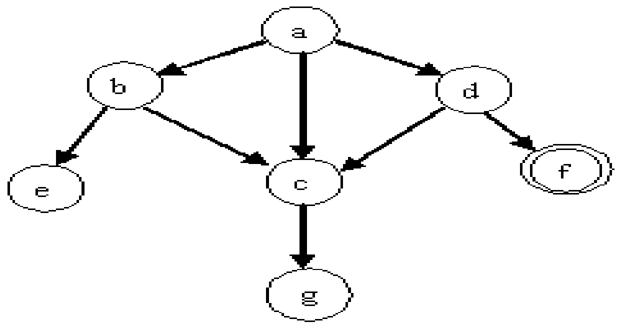

```
/*goal:breadth(['a'],[],'f').
  Open=['a']              Closed=[]
  Open=['b','c','d']      Closed=['a']
  Open=['c','d','e']      Closed=['a','b']
  Open=['d','e','g']      Closed=['a','b','c']
  Open=['e','g','f']      Closed=['a','b','c','d']
  Open=['g','f']          Closed=['a','b','c','d','e']
  Open=['f']              Closed=['a','b','c','d','e','g']
  Goal is found      */
```

Depth First Search

/* A prolog program that applies the concept of depth first search. */

```
%Depth first search program
domains
    c=char.
    l=c*.
predicates
   depth(l,l,c).
   difference(l,l,l).
   append(l,l,l).
   member(c,l).
   print(l,l).
   path(c,c).
clauses
   depth([],_,_):-!,write("Goal is not found ").

   depth([G|T_Open],Closed,G):-!,print([G|T_Open],Closed),write("Goal is found "),nl.
   depth([H|T_Open],Closed,G):-
        print([H|T_Open],Closed),%Print Open & Closed.
        findall(X,path(H,X),Children),%Find children of H.
        append(Closed,[H],Closed1),%Put H in Closed.
        difference(Children,T_Open,Children1),%Ignore children of H if already on Open or
        difference(Children1,Closed1,Children2),%Closed
        append(Children2,T_Open,Open1),%Put remaining children on left of Open.
        depth(Open1,Closed1,G).

   difference([],_,[]):- !.
   difference([H|T],Z,[H|T1]):-
        not(member(H,Z)),!,
        difference(T,Z,T1).
   difference([_|T],Z,T1):-
        difference(T,Z,T1).

   member(H,[H|_]):-!.
   member(H,[_|T]):-
        member(H,T).
```

```
append([],L,L):-!.
append([H|T],L,[H|M]):-
     append(T,L,M).

print(Open,Closed):-
     write("Open=",Open,"        ","Closed=",Closed),nl.

path('a','b').
path('a','c').
path('a','d').
path('b','e').
path('b','c').
path('d','c').
path('d','f').
path('c','g').
/*
goal:depth(['a'],[],'f').
Open=['a']      Closed=[]
Open=['b','c','d']      Closed=['a']
Open=['e','c','d']      Closed=['a','b']
Open=['c','d']      Closed=['a','b','e']
Open=['g','d']      Closed=['a','b','e','c']
Open=['d']      Closed=['a','b','e','c','g']
Open=['f']      Closed=['a','b','e','c','g','d']
Goal is found  */
```



Map Coloring Problem

Prolog Program for the Map Coloring Problem

A simple Prolog program that demonstrates the use of Constraint Satisfaction Problems (CSP) to solve a color problem. In this example, the problem is to color a map of four regions (A, B, C, D) such that no two adjacent regions have the same color. We'll use three colors: red, green, and blue.

```
% Define the colors available
color(red).
color(green).
color(blue).

% Define the constraints
different(red, green).
different(red, blue).
different(green, red).
different(green, blue).
different(blue, red).
different(blue, green).

% Define the CSP for the map coloring problem
   coloring(A, B, C, D) :-
            color(A), color(B), color(C), color(D),
            different(A, B), % A is adjacent to B
            different(A, C), % A is adjacent to C
            different(B, C), % B is adjacent to C
            different(B, D), % B is adjacent to D
            different(C, D). % C is adjacent to D

% Query to find a solution:
            coloring(A, B, C, D).
```

Explanation:

1. Colors Definition:
  - We define the available colors using the `color/1` predicate.

2. Constraints Definition:
  - We define the `different/2` predicate to ensure that two colors are different.

3. CSP Definition:

- The `coloring/4` predicate defines the map coloring problem. It assigns a color to each region (A, B, C, D) and applies the constraints to ensure no two adjacent regions share the same color.

4. Query:
   - The query: coloring(A, B, C, D). % finds a solution to the problem, if one exists.

The Prolog interpreter will return the possible colorings for the regions that satisfy the constraints. For example:

A = red,
B = green,
C = blue,
D = red.