Logic & Constraint Prog.

About Prolog search strategy

Unification

Logic programming emerged from the procedural interpretation of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Unification

Logic programming emerged from the procedural interpretation of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?

Unification

Logic programming emerged from the procedural interpretation of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?
A: Replace the X_is with the t_is in φ_p

Unification

Logic programming emerged from the procedural interpretation of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?
A: Replace the X_is with the t_is in φ_p

Q: What if there are terms in the head of the rule? $p(u_1, \ldots, u_n) \leftarrow \varphi_p$

Unification

Logic programming emerged from the procedural interpretation of logic formulas of the form:

$$p(X_1, \ldots, X_n) \leftarrow \varphi_p$$

which can be read:

"In order to prove $p(X_1, \ldots, X_n)$, it is sufficient to prove φ_p."

Q: What if we try to prove $p(t_1, \ldots t_n)$, where $t_1, \ldots, t_n$ are terms ?
A: Replace the X_is with the t_is in φ_p

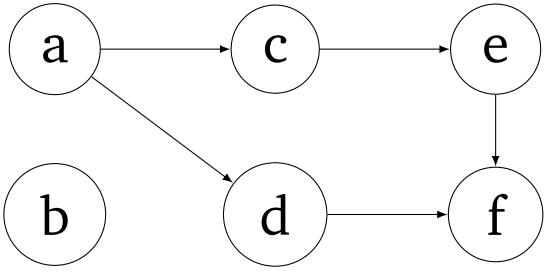
Q: What if there are terms in the head of the rule? $p(u_1, \ldots, u_n) \leftarrow \varphi_p$
A: try to *unify* (match) the u_i's and the t_i's, and make the appropriate substitutions in φ_p

Example $p(X, [X, Y, X])$ can be unified with $p(2, L)$:
$2 \rightarrow X$, $[2, Y, 2] \rightarrow L$.

Non terminating queries

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

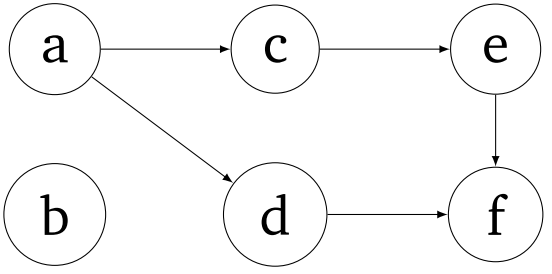
edge(a, c).
edge(a, d).
edge(c, e).
edge(e, f).
edge(d, f).



Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

edge(a, c).
edge(a, d).
edge(c, e).
edge(e, f).
edge(d, f).

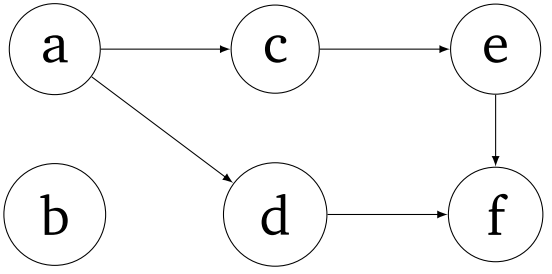


Definition of a predicate path/2, such that path(X, Y) is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
$\mathsf{edge}(X, Y)$ is true if there is an edge between from vertice X to vertice Y.

$\mathsf{edge}(a, c).$ $\quad$ $\mathsf{edge}(a, d).$
$\mathsf{edge}(c, e).$ $\quad$ $\mathsf{edge}(e, f).$
$\mathsf{edge}(d, f).$



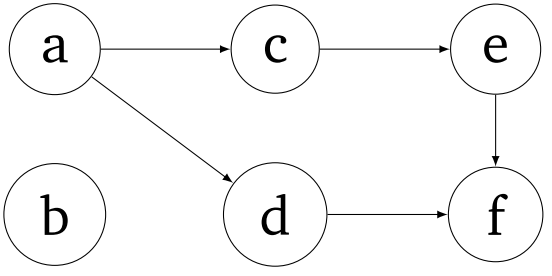
Definition of a predicate path/2, such that $\mathsf{path}(X, Y)$ is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

$\mathsf{path}(X, Y) \leftarrow \mathsf{edge}(X, Y) \vee (\mathsf{edge}(X, Z) \wedge \mathsf{path}(Z, Y))$.

Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

edge(a, c). edge(a, d).
edge(c, e). edge(e, f).
edge(d, f).



Definition of a predicate path/2, such that path(X, Y) is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

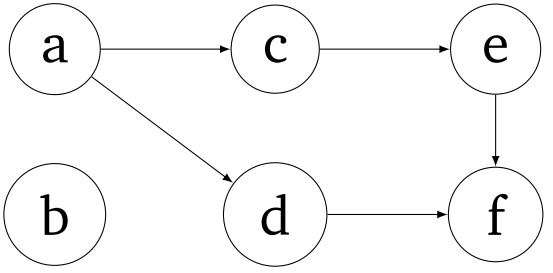
path$(X, Y) \leftarrow$ edge$(X, Y) \vee$ (edge$(X, Z) \wedge$ path(Z, Y)).

Draw the search trees for the following queries:
path(a, e)? path(e, a)? path(a, U)?

Non terminating queries : Retrieval in a graph

Directed graph = binary relation $\Rightarrow$ predicate edge/2:
edge(X, Y) is true if there is an edge between from vertice X to vertice Y.

edge(a, c). edge(a, d).
edge(c, e). edge(e, f).
edge(d, f).



Definition of a predicate path/2, such that path(X, Y) is true if there is a path, of any length, from X to Y:
(Hint: there must an edge from X to Y, or an edge from X to some Z from which...)

path$(X, Y) \leftarrow$ edge$(X, Y) \vee$ (edge$(X, Z) \wedge$ path(Z, Y)).

Draw the search trees for the following queries:
path$(a, e)?$ path$(e, a)?$ path$(a, U)?$

Now, draw the search trees for the query path$(a, f)?$ if we re-define path/2 as follows:
path$(X, Y) \leftarrow$ edge$(X, Y) \vee$ (path$(X, Z) \wedge$ edge(Z, Y)).

Non terminating queries : Retrieval in a graph

Is it better with the following definition?

$\mathsf{path}(X, Y) \leftarrow \mathsf{edge}(X, Y) \vee (\mathsf{edge}(Z, Y) \wedge \mathsf{path}(Z, Y))$.

$\Rightarrow$ In general, it is better to instantiate variables before the recursive call(s).

Prolog negation is not logical negation (again)

Consider the previous graph, but suppose edges mean "attacks" between nodes. We say that a node is safe if:

- it is not attacked at all; or
- it is not attacked by any safe node.

Prolog definition: $\mathsf{safe}(X) \leftarrow \neg(\mathsf{edge}(Y, X) \wedge \mathsf{safe}(Y))$.

Draw the search tree for the following queries:

$\mathsf{safe}(a)?$ $\mathsf{safe}(c)?$ $\mathsf{safe}(f)?$ $\mathsf{safe}(U)?$

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list $\Rightarrow$ Define a predicate path/3 such that path(X, Y, P) is true if P is a list of vertices on a path from X to Y:

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list
$\Rightarrow$ Define a predicate path/3 such that $\mathsf{path}(X, Y, P)$ is true if P is a list of vertices on a path from X to Y:

$$\mathsf{path}(X, Y, P) \leftarrow \mathsf{edge}(X, Y) \wedge P = [\,] \\ \vee(\mathsf{edge}(X, Z) \wedge \mathsf{path}(Z, Y, P1) \wedge P = [Z|P1]).$$

Draw the search tree for the queries $\mathsf{path}(a, e, P)$, $\mathsf{path}(a, b, P)$ and $\mathsf{path}(a, f, P)$.

Recursive predicates that "construct" a list

In some typical applications of graph traversal (e.g. a route planner) one does not only want to check if there is a path from X to Y, but also to *return* that path.
We do not know the length of the path in advance $\Rightarrow$ store it in a list
$\Rightarrow$ Define a predicate path/3 such that $\mathsf{path}(X, Y, P)$ is true if P is a list of vertices on a path from X to Y:

$$\mathsf{path}(X, Y, P) \leftarrow \mathsf{edge}(X, Y) \wedge P = [\,] \\ \vee(\mathsf{edge}(X, Z) \wedge \mathsf{path}(Z, Y, P1) \wedge P = [Z|P1]).$$

Draw the search tree for the queries $\mathsf{path}(a, e, P)$, $\mathsf{path}(a, b, P)$ and $\mathsf{path}(a, f, P)$.

Exercise 1 On the graph example again.

Question 1.1 What happens if we add an edge from c to a?

Question 1.2 Re-define your predicate path/3, so that it is still complete when the graph has cycles.
(Hint: use a list of "forbidden" nodes, and an intermediate predicate path/4.)

Recursive predicates that "construct" a list

Exercise 2 Write definitions for the following predicates to manipulate lists:

select/3**:** a predicate that can be used to "delete" an occurrence of an element of a list. For instance, to the query select$(X, [a, b, c, a], R)$ Prolog should answer:
$X = a \wedge R = [b, c, a] \vee X = b \wedge R = [a, c, a] \vee X = c \wedge L = [a, b, a] \vee X = a \wedge L = [a, b, c]$.

concat/3**:** concat(L, M, R) is true if R is the list that contain the elements of L followed by those of M.

Exercise 3 Define a predicate to compute the length of a list, and anoter predicate to *generate* a list of a given length.

When the result is at a leaf of the Prolog's search tree

It often happens that the result that we want to construct is only available at the leaves of the search tree.
Suppose for instance that we want a predicates that can "reverse" a list:
$\mathsf{reverse}([a, b, c], R) \Rightarrow R = [c, b, a]$
First solution with append: $\mathsf{append}(L_1, L_2, L_3)$ is true if "$L_3 = L_1.L_2$".
$\mathsf{reverse}(L, M) \leftarrow L = [\,] \wedge M = [\,]$
$\quad \vee L = [X|R] \wedge \mathsf{reverse}(R, S) \wedge \mathsf{append}(S, [X], M).$
Number of operations in $O(|L|^2)$.

When the result is at a leaf of the Prolog's search tree

Better solution?

$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$

When the result is at a leaf of the Prolog's search tree

Better solution?

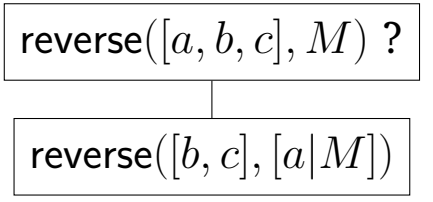
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$

$\mathsf{reverse}([a, b, c], M)$?

When the result is at a leaf of the Prolog's search tree

Better solution?

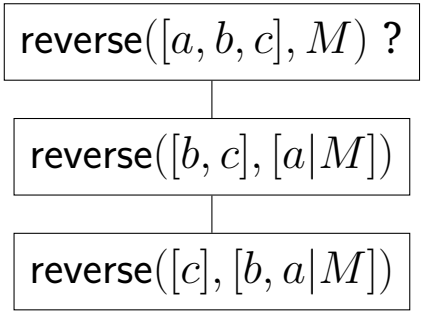
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



When the result is at a leaf of the Prolog's search tree

Better solution?

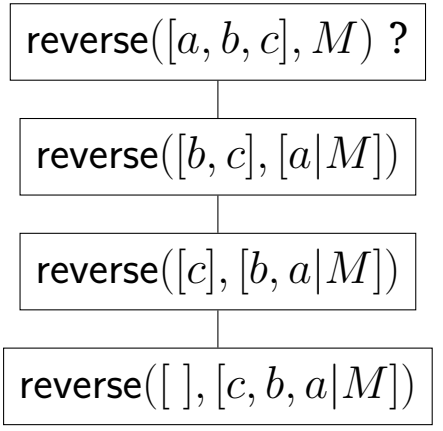
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



When the result is at a leaf of the Prolog's search tree

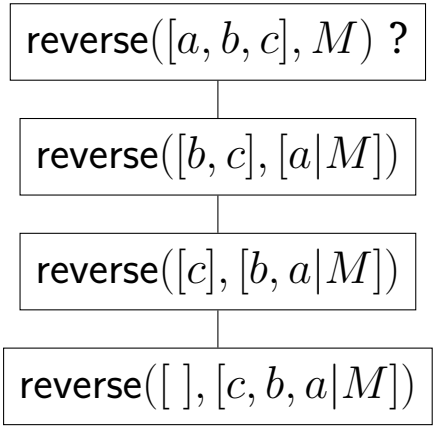
Better solution?

$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



When the result is at a leaf of the Prolog's search tree

Better solution?
$\mathsf{reverse}(L, M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R, [X|M]) \ldots$



With an *accumulator*:
$\mathsf{reverse}(L,A,M) \leftarrow L = [X|R] \wedge \mathsf{reverse}(R,[X|A],M) \vee L = [\,] \wedge M = A$
$\mathsf{reverse}(L, M) \leftarrow \mathsf{reverse}(L, [\,], M)$
Number of operations in $O(|L|)$

Remark The predicates member/2, append/3, select/3 and reverse/2 are usually predefined in Prolog.

List of solutions

Sometimes we would be happy to compute a list of all solutions to a given predicate.
Suppose for instance we want all movies directed by Woody Allen, sorted in alphabetical ordering.

?? Difficult:

- all solutions to the query director$('$*Allen, Woody*$', M)$ are in different branches of Prolog's search tree
- all branches of the search tree are independent of one another

List of solutions

Good implementations of Prolog provide a *meta-predicate* that lists solutions to a given query:

findall$(T, G(T), L)$**:** here $G(T)$ means that G is goal (formula) in which the term T appears; then the call will

- call the goal B;
- for each solution found, instanciate the term T according to the solution;
- construct the list L of these instances T.

For instance, with the movie database excerpt:

findall$(A,$ directed$('$*Fatih Akin*$', A), L)? \Rightarrow L = [$'*Adam Bousdoukos*', '*Mori*

findall$([D, M, Y],$ director$(D, M), L). \Rightarrow L = \ldots$

List of solutions

bagof$(T, G(T), L)$**:** similar to findall, but the results are grouped according to the values of variables of G which do not appear in T

On the example: bagof$(A,$ directed$(D, A), L)? \Rightarrow \ldots$

Exercise 4 Consider the graph above again: define a predicate reachable/2, such that reachable(X, L) is true if L is the list of vertices to which there is a path from X.

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, a recursion necessitates to store in a stack that the successives values of the parameters with which the recursive predicate is called.

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, a recursion necessitates to store in a stack that the successives values of the parameters with which the recursive predicate is called.

For instance, consider the following program, that prints on the screen the integers from X to N:

$\mathsf{count}(X, X).$

$\mathsf{count}(X, N) \leftarrow X < N \wedge \mathsf{write}(X) \wedge Y \,\mathsf{is}\, X + 1 \wedge \mathsf{nl} \wedge \mathsf{count}(Y, N).$

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, a recursion necessitates to store in a stack that the successives values of the parameters with which the recursive predicate is called.

For instance, consider the following program, that prints on the screen the integers from X to N:

$\mathsf{count}(X, X).$

$\mathsf{count}(X, N) \leftarrow X < N \wedge \mathsf{write}(X) \wedge Y \,\mathsf{is}\, X + 1 \wedge \mathsf{nl} \wedge \mathsf{count}(Y, N).$

The query $\texttt{count}(1, 1000000)$ works fine

Last call optimization

One disadvantage of recursion compared to iteration is that, in general, a recursion necessitates to store in a stack that the successives values of the parameters with which the recursive predicate is called.

For instance, consider the following program, that prints on the screen the integers from X to N:

$\mathsf{count}(X, X).$

$\mathsf{count}(X, N) \leftarrow X < N \wedge \mathsf{write}(X) \wedge Y \,\mathsf{is}\, X + 1 \wedge \mathsf{nl} \wedge \mathsf{count}(Y, N).$

The query $\mathtt{count}(1, 1000000)$ works fine

But if we define a predicate to count backwards:

$\mathsf{revcount}(X, X).$

$\mathsf{revount}(X,N) \leftarrow X<N \wedge Y \mathsf{is} X+1 \wedge \mathsf{revcount}(Y,N) \wedge \mathsf{write}(Y) \wedge \mathsf{nl}$

the query $\mathsf{revount}(1,1000000)$ leads to a crash: stack overflow!

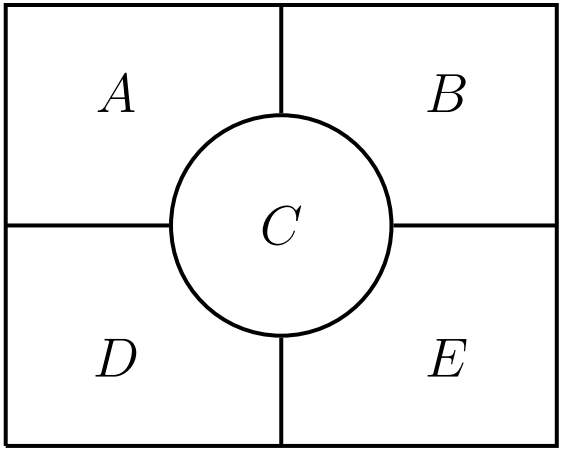
Last call optimization

Explanation Most compilers for functional / logic programming languages are able to transform a recursion into an iteration, if the recursive call is the last one in the rule.

In prolog, the recursion is usually transformed into an iteration if:

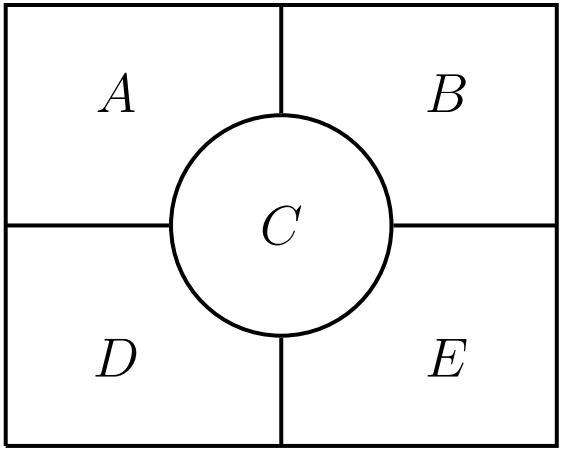
- the recursive call is the last one in the rule, and
- no other rule can be tried after the recursive call
  (the other rules must therefore in general appear *before* the recursive one
- no more backtrack is possible with the other goals in the rule

Generate and test



Choose a color for each region, so that no two adjacent regions have the same color.
3 colors: green, orange, purple
(Is it possible with less colors?)

Generate and test : The "Map coloring" example



Choose a color for each region, so that no two adjacent regions have the same color.
3 colors: green, orange, purple
(Is it possible with less colors?)

$\mathsf{solution}(A, B, C, D, E) \leftarrow \mathsf{generate}(A, B, C, D, E) \wedge \mathsf{test}(A, B, C, D, E).$

$\mathsf{generate}(A, B, C, D, E) \leftarrow$

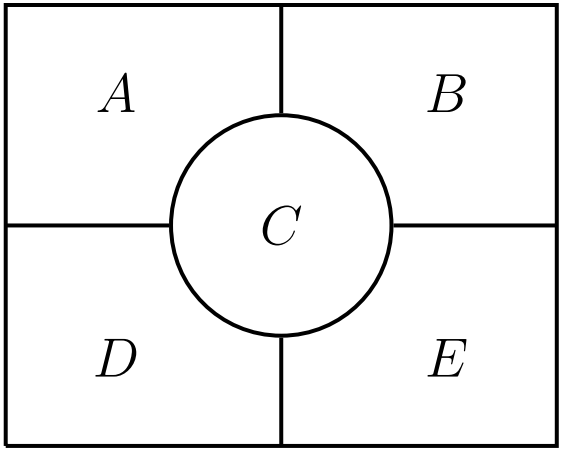
$\quad \mathsf{color}(A) \wedge \mathsf{color}(B) \wedge \mathsf{color}(C) \wedge \mathsf{color}(D) \wedge \mathsf{color}(E).$

$\mathsf{color}(X) \leftarrow X = \mathsf{green} \vee X = \mathsf{purple} \vee X = \mathsf{orange}.$

$\mathsf{text}(A, B, C, D, E) \leftarrow A \neq B \wedge B \neq E \wedge E \neq D \wedge D \neq A$

$\quad \wedge C \neq A \wedge C \neq B \wedge C \neq E \wedge C \neq D.$

Generate and test : The "Map coloring" example



Choose a color for each region, so that no two adjacent regions have the same color.
3 colors: green, orange, purple
(Is it possible with less colors?)

$\mathsf{solution}(A, B, C, D, E) \leftarrow \mathsf{generate}(A, B, C, D, E) \wedge \mathsf{test}(A, B, C, D, E).$
$\mathsf{generate}(A, B, C, D, E) \leftarrow$
$\quad \mathsf{color}(A) \wedge \mathsf{color}(B) \wedge \mathsf{color}(C) \wedge \mathsf{color}(D) \wedge \mathsf{color}(E).$
$\mathsf{color}(X) \leftarrow X = \mathsf{green} \vee X = \mathsf{purple} \vee X = \mathsf{orange}.$
$\mathsf{text}(A, B, C, D, E) \leftarrow A \neq B \wedge B \neq E \wedge E \neq D \wedge D \neq A$
$\quad \wedge C \neq A \wedge C \neq B \wedge C \neq E \wedge C \neq D.$

$\Rightarrow$ How many leaves does the search tree have?
(In general, this problem is called graph coloring.)

Exercises

Exercise 5 Consider the movie database. Let us define the *degree of movie separation* between two actors or actresses A_1 and A_2 as follows: it is 0 if they played in the same movie at least once; it is 1 if they did not play in the same movie, but played in movies that have at least one common actor or actress; it is 2 if it is not 1 and there are two other actors or actresses B_1 and B_2 who played in the same movie, and such that A_1 and B_1 (respectively A_2 and B_2) played in the same movie; and so on... That is, it is the length of the shortest "movie path" between them. By definition, the degree will be infinite if there is no "movie connection" between two persons.

Exercises : On graph traversal

Question 5.1 Define a predicate that can be used to find actors and actresses who have a degree of movie separation of 2 with a given actor or actress A.

Question 5.2 Define a predicate that can compute the degree of movie separation between two given actors or actresses.

Exercises : On graph traversal

Exercise 6 The "flights" prolog database [1] contains Prolog facts with the following information:

- flying times, for instance
  $\mathsf{vol}(\mathsf{it}, 386, \mathsf{blagnac}, \mathsf{cdg}, [14, 30], [15, 30])$
  indicates that flight 386 of the company "it" takes off from Blagnac airport at 14h30 and lands at Charles-de-Gaulle at 15h30; in particular, times of the day are represented using lists giving the hour and minutes.
- flight prices, for instance
  $\mathsf{tarif}(\mathsf{it}, \mathsf{toulouse}, \mathsf{paris}, 500)$
  indicates that a ticket to fly from Toulouse to Paris with "it" costs 500€;
- the cost of airport taxes, for instance
  $\mathsf{taxe}(\mathsf{toulouse}, 100)$
  indicates that every passenger using an airport in Toulouse must pay a 100€tax every time;

[1] www.irit.fr/~Jerome.Mengin/prolog/vols-payants-bd.pl

Exercises : On graph traversal

- location of airports, for instance
  aeroport(toulouse, blagnac)
  indicates that Blagnac airport is near Toulouse.

Question 6.1 Load the file, and submit queries to get: all flight numbers from Blagnac to Orly, flight numbers from Marseille to Paris, all London airports.

Question 6.2 Define a relation directConnection/4 such that connection(A_D
is true if there is a direct flight from airport A_D to airport A_A leaving after time H_D and arriving before H_A.

Question 6.3 Define now a relation route/4 such that route(A_D, H_D, A_A, H
is true if there is a sequence of flights to go from airport A_D to airport A_A leaving after time H_D and arriving before H_A. In case of stopovers, there must be at least 30 minutes if flights arrive and start from the same airport, and 2 hours if one must change airport in the same city.

Question 6.4 Extend the preceding relation so that one can get the price of the ticket, and the itinerary. (Airport taxes are paid for the starting and arrival airports, as well as once for each stopover.)

Exercises : On graph traversal

Exercise 7 Once upon a time a farmer went to the market and purchased a fox, a goose, and a bag of beans. On his way home, the farmer came to the bank of a river and hired a boat. But in crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose, or the bag of the beans. If left alone, the fox would eat the goose, and the goose would eat the beans. The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact.[2] You are asked to write a Prolog program to discover how he did it. More precisely, you must define a predicate that computes a sequence of crossings that leads from the initial state (the farmer and his goods on one side of the river) to the final state (the farmer and his goods on the other side). A state of the problem can be described with a lists of the elements on the initial side of the river. For instance, the initial state could be described by $[\mathsf{f}, \mathsf{g}, \mathsf{x}, \mathsf{c}]$. The main predicate of your program, plan/1, must be defined so that the query:
$?-\mathsf{plan}(L).$

[2] en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle

yields a list of successive states that lead from the initial state to the final state: $L = [\ [\mathsf{f},\mathsf{x},\mathsf{g},\mathsf{c}]\ ,\ [\mathsf{x},\mathsf{c}]\ ,\ [\mathsf{f},\mathsf{x},\mathsf{c}] \ldots\ [\ ]\ ]$.

Question 7.1 Define a predicate safeState/1 such that safeState(E) is true if E represents a state where the fox is not left alone with the goose, and the goose is not left with the corn. (Use the built-in predicate member/2.)

Question 7.2 Define a predicate crossing/2 such that crossing($E1, E2$) is true if it is possible to change from state $E1$ to state $E2$ with a single crossing of the farmer, with or without one good. (Use the built-in predicate select/3.)

Question 7.3 Define a predicate plan/4 such that plan(I, F, P, N) is true if it possible to go from state I to state F with the states in the list P as intermediary states, without going through the states in the list of forbidden states N. (Note: checking that a state is not in N is not trivial, since there are different possibilities to describe one state; for instance, $[\mathsf{f},\mathsf{x},\mathsf{c}]$ and $[\mathsf{x},\mathsf{f},\mathsf{c}]$ may refer to the same state.)

Question 7.4 Finally define plan/1: plan(P) must be true if P is a

sequence of states that describe a valid plan from the initial state to the final state.

Exercises : On list operations

Exercise 8 You are asked to write a program that analyses how some objects are composed of other objects: their components, that can themselves be decomposed into components. A small database contains, for each object, the list of its components, using a relation components/2: components(O, L) is true if L is the list of components of object O.

components$(a, [b, c, d, c])$. components$(b, [e, f])$. components$(f, [g, e$

Thus a has four components, two of type c ; b can itself be decomposed, as well as c and f, whereas d, e, g and h are elementary components.

Question 8.1 Define a predicate allComp/2, such that allComp(O, L) is true if L is the list of all the components, elementary or not, that constitute object O - including O itself. For instance, the query allComp$(a, U$ should yield the answer $U = [a, b, e, f, g, e, c, h, h, h, d, c, h, h, h]$. (The built-in predicate append/3 can be useful.)

Question 8.2 Define a predicate compEl, such that compEl(O, L) is true if L is the list of elementary components of object O. For instance, the query compEl(a, U) should yield $U = [e, g, e, h, h, h, d, h, h, h]$.

Exercises : On the "Generate & test paradigm"

Exercise 9 Ann, Bill, Charlie, Don, Eric own one box each but we don't know which box. We know the size and color of each box: one box is of size 3 and black; one is of size 1 and black; one is of size 1 and white; one is of size 2 and black; the last one is of size 3 and white. We also have some information about the characteristics of the boxes owned by each person :

- Ann and Bill have boxes with the same colour;
- Don and Eric have boxes with the same colour;
- Charlie and Don have boxes with the same size;
- Eric's box is smaller than Bill's.

We want to know who owns which box. In order to solve the problem, you can:

1. Write a Prolog database with the characteristics of the boxes, associating a number to each box, for instance: $\mathsf{box}(2, 1, \mathsf{black})$ represents that the second box is of size 1 and black;
2. Write a predicate to compute the solution of the problem: $\mathsf{solution}(A$ must be true if A is the box owned by Ann, B the one owned by Bill, and so on...

Exercises : On the “Generate & test paradigm”

Exercise 10 The arithmetic cryptographic puzzle: Find distinct digits for S, E, N, D, M, O, R, Y such that S and M are non- zero and the equation SEND+MORE=MONEY is satisfied.