# Introductory notes on Prolog and AI Search

Leonardo Bottaci
Department of Computer Science
University of Hull, Hull, HU6 7RX

## Contents

# 1 Prolog

## 1.1 Introduction

A prolog program consists of facts and rules. The following are some examples of facts.

```
car(rusty).
car(dino).
colour(rusty, blue).
colour(dino, red).
```

These facts state that rusty and dino are both cars. They also state the colour of each car. Notice that a fact has two parts, a property or attribute part such as car or colour and an object part, such as rusty or dino. In prolog, the property is called the functor. The prolog syntax requires that every fact be terminated by a full stop.

A rule is more complex than a fact, roughly speaking, it relates various facts. The following is an example of a rule relating the facts given above.

```
expensive(Thing) :- car(Thing),
                    colour(Thing, red).
```

This rule states that a Thing is expensive (has the property expensive) if the Thing is a car (has the car property) and the Thing is red (has the red property). Notice that :- means if and , means and. A rule has a head, the part before the if, and a body, the part after the if.

Collectively, facts and rules are known as clauses. A collection of clauses constitute a program, i.e.

```
car(rusty).
car(dino).
colour(rusty, blue).
colour(dino, red).
expensive(Thing) :- car(Thing),
                    colour(Thing, red).
```

The above clauses would be written in a file and the file loaded into the prolog interpreter.

When a prolog program executes, it calculates the properties of object, that is all it does. In the above program, some of the properties of some objects can be calculated very easily. For example, the fact that dino has the car property is stated directly in the program. Prolog can be asked to calculate this property of dino as follows. Within the prolog interpreter, type `car(dino).` at the ?- prompt. The ?- prompt is also known as the query prompt.

```
?- car(dino).
```

After typing return, prolog replies

```
Yes
```

It is important to understand in detail how prolog is able to produce the answer Yes in response to the query `car(dino).`. The interpreter reads the query and extracts the functor (property) car and the object dino. It then goes through each line of the program, always starting from the first line to find a fact that has the same functor as the query, i.e. car. Prolog finds the fact `car(rusty).`. The functor of this fact is the same as that in the query but the object does not match. As a result, this fact does not match the query and prolog moves on to the next line of the program. On the next line, it finds `car(dino).` which does match the query and hence prolog replies Yes.

If the query had been

```
?- car(rover).
```

prolog would not have been able to find a match in any of the rows of the program and would therefore have replied

```
No
```

So far, the queries presented have not been very useful. Suppose that the user does not know which objects are cars, i.e. the user cannot enter the query `car(dino).` because although he is interested in the car property but does not know the names of any of the cars in the program. The best query that the user can manage is

```
?- car(X).
```

where X is a variable and stands for any object that has the car property. In prolog, variables are always written with an upper case initial letter. Objects or constants are written with a lower case initial letter. In the prolog rule above, Thing is a variable because of the initial upper case T.

In response to the above query, prolog replies

```
X = rusty
Yes
```

Again, prolog does this by reading the query, extracting the functor and variable. Starting from the first line of the program, the functor in the query is matched against that of the fact or rule in the program. If the functor matches, then the variable of the query is matched against the constant rusty. Since the variable does not yet have a value, it can be given the value rusty and as a result a match is produced. If the variable already had a value, different to rusty then it would not have matched.

It is important to note that a variable without a value may be given a value when there is an attempt to match it. A variable without a value is called unbound. A variable with a value is called bound. So in the above calculation, two steps take place, firstly the unbound variable X is bound to rusty and then the variable, with the value rusty is successfully matched to rusty.

Looking at the program, rusty is not the only car. How can the user find out that dino is also a car? The prolog interpreter allows the user to remove the value of a variable, i.e. to make in unbound, and to force prolog to find another value for the variable that will match. To do this, prolog always remembers at which point in the program a variable was bound to a value. It remembers, for example, that X was bound to rusty at the first line of the program. It the user enters

```
;
```

prolog goes to the point at which the variable X was given the value rusty, removes the value rusty so that X is unbound and moves to the next line of the program so that the same match cannot be made again. The interpreter now finds the fact `car(dino)`. Again, X is unbound and so it can be bound to dino, at which point the query matches the fact and prolog outputs

```
X = dino
Yes
```

It the user once more enters ; in order to ask for a different match, the binding of X to dino is removed and the search for a match starts from the line below where the dino binding was made. No more matches can be found and so prolog outputs No.

Prolog program execution is best understood as a process in which a tree structure of possible matches is generated.

```
   |-- ?car(X),
   |    car(rusty).  X = rusty
   |
   |-- ?car(X),
   |    car(dino).  X = dino
   |
   |-- ?car(X),
        No
```

Whenever, there is a choice about the value that can be bound to a variable, there is another branch in the tree. Prolog execution is best understood by constructing by hand the search tree of matches.

The user can query the program about the colour of various cars. The query `colour(rusty, blue).` will return Yes because the functor and both constants match. The query `colour(rusty, red).` will return No because the functor and only one constant matches. The query `colour(rusty, X).` will return `X = blue` Yes and the query `colour(Y, X).` will return `Y = rusty, X = blue` and if requested, also `Y = dino, X = red`.

Another query the user may enter is `expensive(rusty).`. This query matches `expensive(Thing)` because the variable Thing is unbound and is consequently bound to rusty. Prolog cannot print Yes however, because it does not know that rusty is expensive, it knows only that a Thing is expensive if it is a car and that car is red. Whenever the head of a rule is matched, prolog tries to match the body of the rule.

In the example, `expensive(Thing)` in the program is matched to the query `expensive(rusty).` and so prolog tries to match `car(Thing)` where Thing is bound to rusty. We have worked through this query before. Queries that prolog initiates are called goals. In fact all queries are often called goals. We would say that `expensive(Thing)` is a goal and that `car(Thing)` is a subgoal. To satisfy the subgoal, prolog starts at the first line (as always) and finds the fact `car(rusty).`, i.e. a match because the variable Thing has the value rusty. The rule body has another part that must also be matched. Recall that , means and. The subgoal that prolog now tries to match is `colour(Thing, red).` where Thing is bound to rusty. This subgoal will fail and prolog will print No.

It is possible to construct a tree as before. Given node in tree corresponds to head of rule, children of given node are the subgoals in body of rule.

```
?- expensive(rusty).
   expensive(Thing):- Thing = rusty
   |
   |--- ?car(rusty),
   |     car(rusty).
   |
   |--- ?colour(rusty, red), fail
No
```

Again, it is not necessary for the user to try to guess the names of expensive cars and try each one. The user may enter `expensive(X).`. This query matches `expensive(Thing)`. This raise an interesting situation. There are two variables, X and Thing, neither of which are bound. The rule in prolog is that two unbound variables share. This means that they can be given a binding but only one binding that both of them must share. In effect, when variables share they are made to have equal values, if and when they are given values.

To satisfy the head of a rule, i.e. `expensive(Thing)`, the body of the rule must be satisfied. Prolog now tries to match `car(Thing)` where Thing is unbound but shared with X. We have worked through this query before. Prolog starts at the first line (as always for a new goal) and finds the fact `car(rusty).`, i.e. a match because the variable Thing can be given the value rusty. Recall that Thing shares with X and so X also has the value rusty. The rule body has another part that must also be matched because , means and. The subgoal that prolog now tries to match is `colour(Thing, red).` where Thing is bound to rusty. This subgoal will fail. In the previous example, Thing was bound to rusty because the user had put rusty in the query. Obviously, prolog will not try to replace the user's constant values in a query. In the current example, however, the user query contains a variable X, which prolog currently has bound to rusty. This binding can be undone and a new search made continuing on from where X was given the value rusty.

This is what happens now. Prolog will unbind the last variable binding, in this case, Thing is bound to rusty, and try to find another match for `car(Thing)`. The search for a new match starts from just after the point where the old match was made. A match is found with Thing bound to dino. The next subgoal of the rule body is now rematched, from the first line of the program. This is the subgoal `colour(Thing, red).` where Thing is bound to dino. This subgoal is matched, we also say it succeeds, and when the body of a rule succeeds, so does the head. This is to say that `expensive(Thing)` succeeds and Thing is bound to dino.

The tree of possible matches is

```
?- expensive(X).
   expensive(Thing):- Thing = X, sharing
   |
   |--- ?car(Thing),
   |     car(rusty).  Thing = rusty = X
   |
   |--- ?colour(rusty, red), fail, undo previous match and look for next match
   |
   |--- ?car(Thing),
   |     car(dino).  Thing = dino = X
   |
   |--- ?colour(dino, red),
         colour(dino, red).
X = dino
Yes
```

Exercise 1. Enter the above program into Prolog and execute the queries shown above.

Exercise 2. Modify the above program to add a car called ‘shiny’ with a colour ‘silver’. Any silver car is expensive. Execute by hand, i.e. draw the tree of matches, the query `expensive(X).` to check that `X = dino` and `X = shiny`.

Another example prolog program

```
ring(Person, Number) :-
   location(Person, Place),
   phone_number(Place, Number).

location(Person, Place) :-
   at(Person, Place).

location(Person,Place) :-
   visiting(Person, Someone),
   location(Someone, Place).

phone_number(rm303g, 5767).

phone_number(rm303a, 5949).

at(dr_jones, rm303g).

visiting(dr_bottaci, dr_jones).
```

Exercise 3. Given the above program, execute the queries below. Also enter the program into Prolog check your answers with those of the prolog interpeter.

```
?- location(dr_bottaci, Pl).

?- ring(dr_bottaci, Number).

?- ring(Person, 5767).

?- ring(Person, Number).

?- ring(dr_jones, 999).
```

Exercise 4. Assume that `dr_bottaci` leaves the office of `dr_jones` and instead visits the office of `dr_frankenstein`. Modify the above program to represent this new situation. You will need to make up some room and phone numbers. Test the program.

## 1.2 Syntax

The Prolog data structure is the term. Terms are arbitrarly nested structures defined recursively as:

- (Base case) Constant: Integer, Real, Atom
- (Base case) Variable: named (initial letter upper case), _ (anonymous)
- (recursive case) Structure: functor with parameters or components, each of which may be a term `likes(fred, whisky(malt, 12), X)`

where `likes` and `whisky` are *functors* of *arity* 3 and 2 respectively (an atom is a functor of arity 0). `fred`, `whisky(malt, 12)` and `X` are components of the top level term. Note that full stops are not part of terms.

Prolog has built-in binary operators, so that we may write `3+4*5` rather than `+(3,*(4,5))`. Note that `+(3,*(4,5))` is a structure and not an expression to be evaluated.

Note that syntactically a query has the same form as a fact, the difference is in how they are treated by Prolog. Whereas, the term is a syntactic concept, the clause is a semantic one. A unit clause or fact is a term terminated with a full stop.

```
likes(fred, whisky(malt, 12), X).
```

A non-unit clause, a rule, is constructed from constants and structures with the binary operators "`:-`" and "`,`".

```
descendant(X, Z) :- offspring(X, Y), descendant(Y, Z).
```

A rule, constructed with "`:-`" has a head and a body. Note that there is usually several rules for the same functor. In the rule above, `descendant(X,Z)` is said to be a goal and the two clauses `offspring(X,Y)`, `descendant(Y,Z)` are two subgoals.

Prolog has no global variables; the scope of a variable is a clause invocation.

## 1.3 Matching

Prolog uses rules to try to satisfy queries by:

- Matching the head of a rule;
- Satisfying each subgoal in the body of the rule (from left to right)

If a subgoal within a rule fails, Prolog backtracks and tries to re-match previous subgoals. If a rule fails, Prolog tries to find another matching head (in sequence). Prolog explores choices in order, the backtracking mechanism ensuring that all possibilities are tried, if necessary. Backtracking is an activity under the control of Prolog, not the programmer.

General rules for matching two terms `S` and `T`:

1. if `S` and `T` are constants then they match only if they are the same object
2. if `S` is an unbound variable and `T` is anything, then they match and `S` is bound to `T`.
3. if `S` is a bound variable then the value of `S` is matched with `T`
4. if `S` and `T` are structures then they match only if
   (a) `S` and `T` have the same principal functor, and
   (b) all their corresponding components match

Unification is similar to matching except that it is *illegal* for two components, one being a variable and another being a term, if the variable appears in the term (the *occurs check*). An attempt to match `p(X)` and `p(f(X))` will lead to `f(f(f(f(f(f(....` However, `p(X)` and `p(X)` unify.

## 1.4 Program Semantics

A query is a question about what is logically deducible from the facts and rules present in the program database. Variables in the facts and rules of the program database are universally quantified, i.e.

```
likes(fred, whisky(malt, 12), X).
```

actually means

```
forall X . likes(fred, whisky(malt, 12), X).
```

Variables in a query are existentially quantified, i.e.

```
?- likes(fred, whisky(malt, 12), X).
```

actually means

```
exists X . likes(fred, whisky(malt, 12), X).
```

Clearly if `f(X)` is true for all `X` it is true for some `X`. This is actually a combination of two simpler deduction rules.

```
from   ‘forall X . f(X)’  is deducible    ‘f(a)’   a is anonymous constant

from   ‘f(a)’   a is anonymous constant, is deducible  ‘exists X . f(X)’
```

The result of the query is not simply yes but also the instance of X that has been found to exist. Sometimes the instance must remain anonymous since we may know that there is an instance but not what it is.

A ground term has no variables in it. Think of a non-ground term as denoting a set of ground terms, those ground terms that can be obtained by substituting various ground terms for the variables in the non-ground term. In general, a substitution is a mapping from variables to terms, the terms may contain variables but not any variables that are substituted, i.e. in the domain of the mapping.

```
Y -> car, X -> f(Z, car)
```

Clearly, to produce a ground instance of a non-ground term the substitution must map variables to ground terms. When a substitution is applied (not the use of the word applied, mappings are applied) to a term, the resulting term is called an instance of the original term.

```
Y -> car  applied to  red(Y, X)  produces the instance red(car, X)
```

If one considers the instance term as denoting a set of ground terms, these terms will be a subset of those denoted by the original term.

If t is an instance of s but s is not an instance of t e.g.

```
t = red(car, X)   s =  red(Y, X)
```

then s is said to be more general. Alternatively, set of ground instances available from s contains set of ground instances available from t.

A common instance of two terms is produced by applying the same substitution to the two terms.

```
Y -> car, Z -> 8  applied to red(car, X, Z), red(Y, X, 8) produces red(car, X, 8)
```

The common instance denotes a set of ground terms contained in the intersection of the sets denoted by the terms to which the substitution is applied. Different substitutions produce different common instances. Consider all possible common instances. The most general common instance “contains” all of the other common instances. In face these other common instances are instances of the most general common instance. See e.g. above.

Example: Simulating simple combinations. A half-adder has the logical structure:

```
                    +-----------+
Input1 ----+------->| exclusive |
           |        |    or     |----------> sum
Input2 ----)---+--->|           |
           |   |    +-----------+
           |   |
           |   |    +-----------+
           |   ‘--->|           |
           |        |    and    |----------> carry
           ‘------->|           |
                    +-----------+
```

Represent the behaviour of the devices.

```
xorGate(0,0,0).
xorGate(1,0,1).
xorGate(0,1,1).
xorGate(1,1,0).

andGate(0,0,0).
andGate(0,1,0).
andGate(1,0,0).
andGate(1,1,1).

halfAdder(Input1,Input2,Sum,Carry) :-
   xorGate(Input1,Input2,Sum),
   andGate(Input1,Input2,Carry).
```

Exercise 5. Enter the above program and execute the queries `?- halfAdder(0,1,S,C).` and `?- halfAdder(1,1,S,C).`.

Exercise 6. A full adder can be made as follows:

```
                        +-------+
carry ----------------->|       |------------->sum
in                      | half  |
           +-------+    | adder |    +----+
Input1 --->|       |--->|       |--->|    |
           | half  |    +-------+    | or |--->carry
           | adder |                 |    |    out
Input2 --->|       |---------------->|    |
           +-------+                 +----+
```

Modify the code given above by adding code for the predicates `orGate(input1, input2, output).` and `adder(Input1, Input2, CarryIn, Sum, CarryOut).` and execute the queries `adder(1,0,1,Sum,Carry).` and `adder(1,1,1,Sum,Carry).`.

## 1.5 Recursion

No loops in Prolog, must use recursion.

```
factorial(1, 1).
factorial(N, Fac):- M is N - 1,
                    factorial(M, NewFac),
                    Fac is NewFac * N.
```

Note that `is`, `-` and `*` are three infix functors. This is syntactic sugar, imagine that `M is N - 1` is immediately rewritten to `is(M, -(N, 1))`. `is` is a built-in predicate but it could in principle be defined by a large collection of facts of the form

```
is(0, -(2, 2)).
is(1, -(2, 1)).
is(2, -(2, 0)).
```

Again, execution is best understood by enumerating the search tree. Construct tree as each rule instance invoked. Tree node corresponds to head of rule, children of node are subgoals in body of rule.

```
?- factorial(3, X).
   factorial(N, Fac):-  N = 3, Fac = X
   |
   |--- ?M is N - 1,  M = 2
   |
   |--- ?factorial(2, NewFac),
   |      factorial(N, Fac):-  N = 2, Fac = NewFac
   |      |
   |      |--- ?M is N - 1,  M = 1
   |      |
   |      |--- ?factorial(1, NewFac),
   |      |      factorial(1, 1).  1 = NewFac
   |      |
   |      |--- ?Fac is NewFac * N.   Fac = 2
   |
   |---  ?Fac is NewFac * N.   Fac = 2 * 3 = 6
```

Note that there is no concept of input or output parameters or arguments in a query. There is, however, the important distinction between bound and unbound variables in the query. Bound variables supply values, unbound variables acquire values when they are bound. In prinicple, it should be possible to "run the factorial function backwards" by executing the following query

```
?- factorial(X, 6).
```

In practice this will not work because of the way in which `is` is defined, i.e. for efficiency.

Exercise 7. Define `myis`, using explicit facts as described above, for a few numbers and rewrite `factorial` to use `myis` then execute `?- factorial(X, 6).`

## 1.6 Lists in Prolog

A list is a sequence of terms (of any length). The list is the important dynamic data structure. Recall the need for dynamic data structures to implement AI problem solving algorithms. A list is in fact a binary tree structure.

```
.(Head, Tail)
```

where `Head` is a term and `Tail` is a list. This notation is difficult to read and so lists are constructed using '`|`' operator with an element and a given list. In addition elements of a list are enclosed in square brackets and separated by commas. The empty list is written `[]` and is a constant atom. Prolog is a mono-typed language, there is one type, the term. A list is nothing more than a term with a particular structure.

```
[peach,2,b,[3,4,5],colour(dino, red)]
```

Lists are manipulated by matching the head and the tail to different variables:

```
[H|T]
[1|[2,3,4,5]]
[a|[b,c,d]]
[peach|[pear,plum]]
```

The `[H|T]` notation can be used to form new lists. Note the use of the = built-in operator that attempts to match two terms.

```
?- Z = [5|[4,3,2,1]].
Z = [5,4,3,2,1]
yes
?- Z = [5,4,3,2,1|[]].
Z = [5,4,3,2,1]
yes
```

The `[H|T]` notation can also be used in the form `[First,Second|Tail]` to separate the first two elements from the rest.

```
?- [F,S|T] = [5,4|[3,2,1]]
S = 4
T = [3,2,1]
F = 5
```

The following pairs are equivalent.

```
[fred,ann,mary,joe]     [fred|[ann,mary,joe]]
[a]                     [a|[]]
```

## 1.7 Accessing elements of a list

Lists are recursive structures therefore access is done by recursion. Access the head and then recurse on the tail. For example, to write out the elements of a list, one element per line, in order.

```
writelist([]).
writelist([H|T]):-write(H), nl, writelist(T).

?- writelist([5,4,3]).
   writelist([H|T]):-  H = 5, T = [4,3]
     ?write(5),
     ?nl,
     ?writelist([4,3]).
      writelist([H|T]):-  H = 4, T = [3]
        ?write(4),
        ?nl,
        ?writelist([3]).
         writelist([H|T]):-  H = 3, T = []
           ?write(3),
           ?nl,
           ?writelist([]).
            writelist([]).

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

append([], L, L).
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
```

Reading in a list from the keyboard

```
getlist([H|T]):-read(X), not X=end, H=X, getlist(T).
getlist([]).

?- getlist(Mylist).
|: fred.
|: ann.
|: mary.
|: end.
Mylist = [fred,ann,mary]
```

Note the use of the `not` predicate. This predicate is a prefix operator, for readability. The query `not(X).` will succeed whenever `X` cannot be matched. Note that

```
not membber(1, [1, 2, 3]).
```

will return `yes` because the misspelt `membber` cannot be matched. Beware also of absent code causing failures.

Exercise 8. Work through Tutorial 2. Tutorial 2 is accessible from the 'Prolog Tutorials' link on the module home page.

## 1.8 Abstract Data Types in Prolog

The basic facility in Prolog for building data structures is the structure. Some versions of Prolog provide modules for information hiding but we will not consider them. We will consider only the functional abstraction that is necessary to implement ADT's.

We will use a list to represent a Stack, i.e. a LIFO queue.

```
stackEmpty([]).
stack(Top, [Top|StackLessTop], StackLessTop).
```

Note that the push, pop and top operations are all available via a single clause depending on the bindings of the variables. The query

```
?- stack(T, [a,b,c], SLT).
```

will bind `T` to `a` and `SLT` to `[b,c]`. The query

```
?- stack(cat, S, [dog,rabbit,horse]).
```

will bind `S` to `[cat,dog,rabbit,horse]`.

A queue has an implicit FIFO discipline. Again we will represent the queue as a list to which elements are added to the end and removed from the front.

```
emptyQueue([]).
enqueue(Item, Queue, NewQueue) :- append(Queue, [Item], NewQueue).
dequeue(Item, [Item|NewQueue], NewQueue).
```

The comments about the binding a variables for the `stack/3` relation also apply to `dequeue/3` and to `enqueue/3`. It is often convenient construct a queue from a single given element.

```
mkqueue(Item, Queue) :- emptyQueue(EmptyQueue),
                        enqueue(Item, EmptyQueue, Queue).
```

Notice that the call to `emptyQueue/1` with the unbound variable binds this variable to the empty queue and thus `queueEmpty/1` serves for both the functions `queueEmpty :: Queue a -> Bool` and `emptyQueue :: Queue a`.

# 2 Basic AI problem solving

## 2.1 Introduction to state space search

Problem solving is a form of thinking and a form that is more easily studied. There are two basic kinds of problem solving, problem solving by use of a systematic procedure or algorithm and problem solving by systematic trial and error or search. The multiplication of two five digit numbers is done by a systematic procedure. A solution to x + y = 23 can be found by systematic trial and error. A systematic procedure is usually used in preference to systematic trial and error. However, for many problems, there may be no known solution procedure or it may involve too many steps and hence be too time or space consuming to use.

Systematic trial and error is used extensively in numerical analysis. However, it use there is to solve numerical problems. Many problems are not numerical, these are known as symbolic. An example of a symbolic problem is the 8-tile puzzle, the diagram below shows the various tile configurations that might be explored during systematic trial and error or search.

```
                       1  5  2
                       7     6      initial state
                       8  4  3
        ______________/ /   \ \_____________
       /               /     \              \
   1     2      1  5  2      1  5  2      1  5  2       next state
   7  5  6         7  6      7  4  6      7  6           or node
   8  4  3      8  4  3      8     3      8  4  3
                                            /
    north        west        south         / east              move
                                          /
                                      1  5  2
                                      7  6  3
                                      8  4
                                    /
                                   .   south
                                  .
                                 /
                          1  2  3
                          8     4      goal state
                          7  6  5
```

As another example, consider the problem of rearranging a stack of 3 blocks,

```
  |---|                                        |---|
  | c |                                        | a |
  |---|    rearrange to the stack              |---|
  | b |                                        | b |
  |---|                                        |---|
  | a |                                        | c |
  |---|                                        |---|
```

Only a single block can be moved at a time, i.e. cannot remove or insert block into the middle of the stack.

Many problems can be tackled by systematic trial and error including, finding an object in a room, diagnosis of a fault in some equipment or of a disease, proving theorems, finding solutions to prolog programs and so.

In order to solve a problem using systematic trial and error or search, the problem must be formulated in a suitable way. It must be possible to identify each of the following.

- A set of all possible states or situations (nodes), the state space. These are the potential solutions that are to be explored. A space is a set with some kind of structure imposed on it. The structure is the relationships or moves between the individual states.

- A set of legal moves from state to state. Not all moves apply to all states. For example, only a single block can be moved in the block stacking example. Only a tile next to the black can be moved in the 8-tile problem.
- A starting state or node.
- A goal state or node, i.e. the solution.

Once a problem is formulated in the above terms, a program can be written to perform the search. For example, reconsider the problem of rearranging a stack of 3 blocks,

Each arrangement of blocks is a state and needs to be represented as a data structure. Each stack is a list, we have a list of three stacks because each block might at some stage be in its own stack.

```
[[c, a, b], [], [],]              start state

[[a, b, c], [], [],]   [[], [a, b, c], [],]  [[], [], [a, b, c]]   end state
```

Note that in our representation the three stacks are represented in a list which is sensitive to the order of the elements. We do not care about the order of the stacks and so the single final state is represented by any one of three lists. We could also be similarly abstract about the initial state.

## 2.2 Graphs in a logic language

We give a possible implementation for graphs in Prolog. In this implementation the graph does not change during the execution of the program, it is not dynamic. This allows us to define each edge by a (unit) clause.

```
edge(a, g).
edge(b, c).
edge(b, d).
edge(c, d).
edge(c, e).
```

If the graph edges are undirected then we add the rule

```
edge(V1, V2) :- edge(V2, V1).
```

If we want to associate a cost with each edge we can define `edge` differently.

```
edge(a, g, 6).
edge(b, c, 5).
edge(b, d, 2).
edge(c, d, 0).
edge(c, e, 6).
```

The predicate `findall/3` constructs a list (the third parameter) of all the possible bindings of a variable (the first parameter) in a predicate (the second parameter) that is satisfied as many times as possible, e.g.

```
?- findall(Node1, edge(Node1, Node2), NodeList).
NodeList = [a, b, b, c, c]
```

if the rule `edge(V1, V2) :- edge(V2, V1).` is not present. Note that duplicates arise from different matches but these can easily be removed.

Exercise 9. What is the value of `NodeList` if the graph is undirected?

The following counts the number of edges that connect to a given node.

```
inDegree(Node, N) :- findall(Node1, edge(Node1, Node), NodeList),
                     removeDuplicates(NodeList, M).
                     length(M, N).
```

Exercise 10. Define the predicate `removeDuplicates(NodeList, M).` Hint, use the `member` predicate.

A dynamic graph can be implemented as a list of edges. Each edge is represented as a structure.

```
edge(Node1, Node2)
```

Note how structures are created in the same way that lists are, i.e. you simply write the term which may contain variables. We prefer a structure rather than a list of two nodes because the structure enforces the constraint that an edge in the graph is represented by exactly two nodes. Given a list of list pairs, we assemble a list of edges to make the graph. Note that we have already seen code to read a list of atoms.

```
mkGraph([], []).
mkGraph([[Node1, Node2]|NodePairs], [edge(Node1, Node2)|Graph]) :-
   mkGraph(NodePairs, Graph).
```

Exercise 11. Write out the execution trace of `?- mkGraph([[a,g], [b,c]], G).`.

The following predicate determines if two nodes are connected by an edge in a given graph.

```
inGraph(Node1, Node2, Graph) :- member(edge(Node1, Node2), Graph).
```

The list of nodes that are connected to a given node in a given graph.

```
adjacent(Node, Graph, Nodes) :-
  findall(AdjNode, inGraph(Node, AdjNode, Graph), Nodes).
```

A more direct recursive definition is based on the rule that the list of adjacent nodes is formed by adding an adjacent node to the list of adjacent nodes in the graph formed once the edge connecting the given node to the adjacent node has been removed. If this edge is not removed the list of adjacent nodes will contain a single node repeated many times.

```
adjacent(Node, Graph, [AdjNode|AdjNodes]) :-
  inGraph(Node, AdjNode, Graph),
  removeEdge(Node, AdjNode, Graph, NewGraph),
  adjacent(Node, NewGraph, AdjNodes).
```

Exercise 12. define `removeEdge/4`? Hint use the predicate for removing a given element from a list.

The presence of a functional interface to the graph data type means that there is no need to represent the graph using data structures at all. It is important only that the various predicates can always be satisfied. This approach is the typical one when dealing with very large graphs, as we will be in this module. The key to defining such a graph is to define

```
inGraph(Node, AdjNode) :- successor(Node, AdjNode).
```

where `successor/2` depends on the details of the graph to be defined. Various examples are encountered in the rest of this module.

## 2.3 Depth first search

In depth first search the first successor is always chosen. If the current node is not the goal node then we try the first successor, there is in general more than one successor. Our search for the goal node will lead to a sequence or path of nodes, each the first successor of the previous node. To construct a list of nodes from the initial node to the goal node we use the recursive relation; the list from the current node to the goal node is equal to the list obtained by adding the current node to the list of nodes from the successor of the current node to the goal node.

```
   Node ------> Node1 ------> ... --------> Goal Node
```

```
dfsA(Node,        %INITIAL NODE
     [Node]) :-   %LIST OF NODES TO GOAL NODE
   goal(Node).

dfsA(Node, [Node | NodesToGoal]) :- successor(Node, Node1),
                                    dfsA(Node1, NodesToGoal).
```

A problem with always choosing the first successor is that of looping, i.e. we repeatedly visit the same set of states. A technique for avoiding looping is to add cycle detection. We keep a list of all the nodes visited so far, as each successor node is generated it is compared with the nodes visited so far. So, taking the `dfsA` predicate above, it is easily extended to avoid looping.

```
dfsB(Node, Solution) :- dfsB3(Node, [], Solution).

dfsB3(Node,                % INITIAL NODE
      History,             % NODES VISITED SO FAR
      [Node | History]) :- % History IS PATH TO GOAL IN REVERSE ORDER
   goal(Node).

dfsB3(Node, History, Solution) :-
   successor(Node, Node1),
   not member(Node1, History),
   dfsB3(Node1, [Node | History], Solution).
```

Note that in the above clause for `dfsB`, the variable `Solution` is simply carried along unbound until goal node is found at which point it is bound to the list of node visited so far. We do this in order that the `History` list may be reversed or otherwise modified before being bound to `Solution`.

Exercise 13. Why is the technique of using a history list for cycle detection not used by operating systems to detect programs that are stuck in a non-terminating loop?

In addition to the problem of looping there is the problem of the very deep path. Suppose the initial state is a very tall stack of blocks and the final state is any state that contains the stack

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

The program might unstack blocks from the tall stack for a long time without repeating any state. This may happen even though the blocks necessary to build the solution stack `a`, `b`, and `c`, may occur close to the top of the very tall stack. To avoid this problem we can limit the depth of recursion, i.e. the number of first successors examined. Add depth count to limit very deep paths. A depth limit will eventually terminate looping.

Take program `dfsA` and add additional variable to count depth.

```
   %IF CURRENT NODE = GOAL NODE, REQD LIST CONTAINS JUST THIS NODE
dfsC(Node, [Node], _) :- goal(Node).
   %ELSE ADD CURRENT NODE TO LIST FROM SUCCESSOR TO GOAL
dfsC(Node, [Node|NodesToGoal], Depth) :-
   Depth > 0,
   successor(Node, Node1),
   NewDepth is Depth - 1,
   dfsC(Node1, NodesToGoal, NewDepth).
```

Once the predicate `dfsC` is called with an argument of 0, it will fail. Backtracking initiates another call of this predicate with a different successor, if one exists. This other call, however, still has the same 0 argument and so will also fail. Indeed, the predicate `dfsC` will fail for all successors at that depth and so backtracking causes a return to the previous depth to explore successors at that level.

Problem we now have is to chose a value for `Depth`. Can repeatedly call `dfsC` with successively higher values of `Depth`. The states close to the initial state are recomputed each time but this is not too inefficient because most of the work is done at the deepest levels of the search because there are more states to compute at these levels due to fan out. We assume that the search space fans out so that by doubling the depth say, the number of states included is much more than than double.

Exercise 14. Write a depth first search program in Prolog which can perform an acyclic depth limited search of a graph.

## 2.4 Rearranging stacks problem using depth first search

The set of state transitions in the stack problem is more conveniently represented by a `successor/2` predicate that generates them on demand. The relation

```
successor(state1, state2)
```

is true if by moving the top of a stack it is possible to move from stacks in state1 to stacks in state2. We are allowed to place a block, only on the top of a stack. To define `successor/2` we need to choose some arbitrary stack, take its top and place it on some other arbitrarily chosen stack. In the following program we use the predicate `del/3` which deletes, and therefore picks, some arbitrary element of a list.

```
  %EG del(b, [a, b, c], [a, c]) -- b DELETED FROM LIST [a, b, c] IS LIST [a, c]
del(X, [X|L], L).                         %DELETE THE FIRST ELEMENT
del(X, [Y|L], [Y|L1) :- del(X, L, L1).    %DELETE A NON-FIRST ELEMENT

successor(Stacks, [Stack1, [Top1 | Stack2] | OtherStacks]) :-
     %Stacks IS BOUND, ONCE Stack1, Top1, Stack2 AND OtherStacks ARE
     %BOUND THEN THE NEXT STATE CAN BE CONSTRUCTED.
   del([Top1 | Stack1], Stacks, StacksLess1),     %ISOLATE A STACK AND ITS TOP
   del(Stack2, StacksLess1, OtherStacks).         %ISOLATE ANOTHER STACK
```

Note that the above predicate will work with any number of stacks from two upwards. The goal condition is

```
goal(State) :- member([a, b, c], State)
```

## 2.5 Generate and test for finding successors

The definition of the `successor/2` predicate is responsible for generating a legal successor state. It is often useful to break this down into a number of steps. For example, it is sometimes more convenient to generate a legal successor state by generating potential successor states and testing each state for legality.

```
    Node  ----> move1 ----> Node1 test
    Node  ----> move2 ----> Node2 test    etc until a Nodei passes the test.
```

A move transforms a node into another node. A move differs from a successor relation in that a move may produce an illegal state or node for a given node and thus the nodes produced may not all be valid successors of the given node. Recall that a successor relation defines exactly those states that are valid successors.

Consider the Towers of Hanoi problem. A ring can be moved from one peg to another. The moves are `move12`, (move a ring from peg 1 to peg 2), `move13`, `move21`, etc. Some moves, however, will be illegal in a given state since a ring must not be placed above a smaller ring.

Each peg is represented by a stack, rings are represented by numbers.

```
[[1, 2, 3], [], []]  is the initial state or node.

[[], [], [1, 2, 3]]  is the final state or goal node hence,

goal([[], [], [1, 2, 3]]).
```

The moves are represented by the `move/2` predicate. The `move(Node, aMove)` goal succeeds if `aMove` is a possible but not necessarily legal move for `Node`. We need check only that a move is from a non-empty peg. We can do this by ensuring that the list bound to that peg in non-empty, i.e. that the list matches the pattern `[H|T]`.

```
move([[Top1 | Stack1]|OtherStacks], move12).
move([[Top1 | Stack1]|OtherStacks], move13).
move([Stack1, [Top2 | Stack2]|OtherStacks], move21).
move([Stack1, [Top2 | Stack2]|OtherStacks], move23).
move([Stack1, Stack2, [Top3 | Stack3]|OtherStacks], move31).
move([Stack1, Stack2, [Top3 | Stack3]|OtherStacks], move32).
```

For each move, a new node must be constructed. The `update(Node, aMove, Node1)` goal succeeds if `Node1` is the result of applying `aMove` to `Node`.

```
update([[Top1 | Stack1], Stack2 | OtherStacks], move12,
      [Stack1, [Top1 | Stack2] | OtherStacks]).

update([[Top1 | Stack1], Stack2, Stack3 | OtherStacks], move13,
      [Stack1, Stack2, [Top1 | Stack3] | OtherStacks]).

update([Stack1, [Top2 | Stack2], Stack3 | OtherStacks], move21,
      [[Top2 | Stack1], Stack2, Stack3 | OtherStacks]).

update([Stack1, [Top2 | Stack2], Stack3 | OtherStacks], move23,
      [Stack1, Stack2, [Top2 | Stack3] | OtherStacks]).

update([Stack1, Stack2, [Top3 | Stack3] | OtherStacks], move31,
      [[Top3 | Stack1], Stack2, Stack3 | OtherStacks]).

update([Stack1, Stack2, [Top3 | Stack3] | OtherStacks], move32,
      [Stack1, [Top3| Stack2], Stack3 | OtherStacks]).
```

A node is legal if every ring is above a larger ring. We need only test the top two rings since if these are never illegal then lower rings, which must have once been at the top, can never be in an illegal order.

```
legal([Peg1, Peg2, Peg3]) :-
   legalTower(Peg1),
   legalTower(Peg2),
   legalTower(Peg3).

legalTower([]).
legalTower([A]).
legalTower([A, B|Rest]) :- A < B.
```

We can now assemble all the above elements into a depth first search program. Obviously we need cycle detection because there is the possibility that a ring will be moved back and forth between the same two pegs. No depth limit is strictly required since the space is not infinite. We may, however, impose a depth limit if we know the depth of a solution.

```
   %Moves IS THE LIST OF MOVES TO SOLVE THE PROBLEM FROM STATE State
solve(State, Moves) :- dfsD(State, [State], Moves).

   %NO MOVE NECESSARY AT THE GOAL STATE
dfsD(State, History, []) :- goal(State).

   %ELSE MAKE A MOVE, IF LEGAL RECURSE FROM THE NEW STATE, ADD MOVE TO LIST
dfsD(State, History, [Move|Moves]) :-
   move(State, Move),
   update(State, Move, State1),
   legal(State1),
   not member(State1, History),
   dfsD(State1, [State1|History], Moves).
```

See also wolf, goat, cabbage problem. Shapiro *The Art of Prolog* p286.

Exercise: Consider a simple timetabling problem in which a set of lecturers are to be assigned to a set of slots (one lecturer per slot). A lecturer may teach in more than one slot but may not teach two consecutive slots. Consider other constraints such as a limit to the total number of slot a lecturer may teach, specific lecturers may teach only in specified slots, and so on. Hint: represent state as a list of three lists, a list of lecturers, a list of unassigned slots and a list of lecturer slot pairs. Use a similar approach to that used for the towers of Hanoi.

## 2.6 Breadth first search

Breadth first search is easily implemented using a FIFO queue. The successors of a given node are added to the end of the queue. Nodes are visited from the front of the queue.

```
bfs(Origin, Visited) :- bfs3([Origin],  % QUEUE
                             [],        % LIST OF NODES VISITED SO FAR
                             RevVisited),  % SOLUTION LIST OF NODES
                        reverse(RevVisited, Visited).

bfs3([Node|_], History, [Node|History]) :- goal(Node).

bfs3([Node|RestQ], History, RevVisited) :-
   not goal(Node),
   findall(NextNode,
           (successor(Node,NextNode),
              not member(NextNode, History),
              not member(NextNode, RestQ)),
           Successors),  %LIST OF SUCCESSORS OF Node
   append(RestQ, Successors, Queue),    %MAKE NEW QUEUE
   bfs3(Queue, [Node|History], RevVisited).
```

Comparing breadth first and depth first, we notice that the depth first program in Prolog does not require a stack because we can use the stack maintained by Prolog for backtracking. Clearly, there is no overall efficiency gain in not using an explicit stack because of the cost of the Prolog maintained stack. The rate of queue growth depends on the average number of successors at each node.

The advantage of using a queue is that other search strategies can be obtained by altering the order in which the successor nodes are maintained on the queue. If children are added to the front, we obtain depth first.

## 2.7 Logic language implementation of 8-tile problem

Each node is a configuration of tiles, i.e. a list of tile positions. List contains x/y coords of each tile in ascending order. Coords of i-th item in list is position of i-th tile. The first tile in the list is the blank. 1/1 is lower left corner of board. 1/2 is simply a structure with an infix functor and a convenient way of grouping two values. Alternatively, a structure like ‘position(1, 2)’ could be used. This would be matched against position(X, Y) to extract each of the two values. Infix operators (which are in reality functors) can be defined by `op(precedence, xfy, /).`

```
goal([2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]).
   %blank at 2/2, tile 1 at 1/3, etc

%successor OF [Empty | Tiles] IS [Tile | Tiles1] WITH COST 1
%ONE OF THE TWO TILES THAT MOVES MUST BE THE BLANK
successor([Empty | Tiles], [Tile | Tiles1], 1) :-
   %SWAP Empty WITH  Tile IN Tiles TO GET Tiles1
   swap(Empty, Tile, Tiles, Tiles1).

%SWAP Empty WITH FIRST TILE
swap(Empty, Tile, [Tile | Ts], [Empty | Ts]) :-
   %MANHATTAN DISTANCE OF 1 (IE ADJACENCY) ENSURES SWAP IS LEGAL
   mandist(Empty, Tile, 1).

%SWAP Empty WITH NON-FIRST TILE
swap(Empty, Tile, [T1 | Ts], [T1 | Ts1]) :-
   %SWAP Empty WITH FIRST TILE IN TAIL
   swap(Empty, Tile, Ts, Ts1).

%MANHATTAN DISTANCE BETWEEN TWO TILES IS THE x + y DISTANCE
mandist(X/Y, X1/Y1, D) :-
```

```
   dif(X, X1, Dx),
   dif(Y, Y1, Dy),
   D is Dx + Dy.

dif(A, B, D) :-
   D is A - B,
   D >= 0.

dif(A, B, D) :-
   D is B - A,
   D > 0.
```

Example using a 2x2 board.

```
?- successor([1/2, 1/1, 2/2, 2/1], N, C).

| ?- successor([1/2, 1/1, 2/2, 2/1], N, C).
   (  1)  1 call: successor([1/2, 1/1, 2/2, 2/1], _23, _24) ?
   (  2)  2 call: swap(1/2,  _47, [1/1, 2/2, 2/1], _48) ?
   (  3)  3 call: mandist(1/2, 1/1, 1) ?
   (  3)  3 exit: mandist(1/2, 1/1, 1)
   (  2)  2 exit: swap(1/2, 1/1, [1/1, 2/2, 2/1], [1/2, 2/2, 2/1])
   (  1)  1 exit: successor([1/2, 1/1, 2/2, 2/1], [1/1, 1/2, 2/2, 2/1], 1)
```

## 2.8 N-queens problem

The problem is to place N queens on an N by N board in such a way that no queen may take any other queen. Think of this problem as a prototype for resource allocation problems. We use a list to represent a partial solution

```
[Col1/Row1, Col2/Row2, ...]
```

where each item in the list is the position of one of the queens. In the goal state the list has eight items (assume N = 8). A state is represented by a partial solution and the proposed column position for the next queen. We use a structure with the functor `node`.

```
node(ProposedCol, [Col1/Row1, Col2/Row2, ...])
```

The successor relation extends the partial solution by using the proposed column and picking an arbitrary row. The next column becomes the new proposed column.

```
successor(node(Col, [PartSoln]), node(NextCol, [Col/Row |PartSoln])) :-
  NextCol is Col + 1,
  member(Row, [1, 2, 3, 4, 5, 6, 7, 8]).
```

We require a predicate to test the validity of a given state, i.e. check that the queens do not threaten each other. Since threaten is a symmetric relation, we need only check that the last placed queen does not threaten any previously placed queen.

```
valid(node(_, [Col1/Row1])).

valid(node(_, [Col1/Row1, Col2/Row2, | RestSoln])) :-
   safe(Col1/Row1, Col2/Row2),
   valid(node(_, [Col1/Row1 | RestSoln])).

safe(Col1/Row1, Col2/Row2) :-
   not (Col1 = Col2),
   not (Row1 = Row2),
   Ldiag1 is Col1 - Row1,    %LEADING DIAGONAL FOR QUEEN 1
   Ldiag2 is Col2 - Row2,    %LEADING DIAGONAL FOR QUEEN 2
   not (Ldiag1 = Ldiag2),
   Tdiag1 is Col1 + Row1,
   Tdiag2 is Col2 + Row2,
   not (Tdiag1 = Tdiag2),
```

The goal node has a proposed column value greater than the number of queens.

```
goal(node(Col, [Col1/Row1 | RestSoln])) :- Col > 8.
```

The following is a solution that is based on selecting values from a small domain of possible values. This is the approach used for certain kinds of finite domain constraint satisfaction problems. The solution shown here, uses the method known as domain reduction. Each variable has an associated domain, the set of values it may take. If for any variable, this set is empty, no solution exists. If this set contains exactly one member, the variable is bound to that value. Initially, the domains will contains more than one member. Briefly, domain reduction is done as follows. A value for a variable is chosen from the variable's domain, reducing that domain to a single value. This value is substituted into the constraints and as a result may reduce the size of the domains of other variables. Repeat this until all variables are assigned or a domain becomes empty.

```
queens(Ycoords) :-
  queens(Ycoords,              %SOLUTION, BOUND ON COMPLETION
         [1,2,3,4,5,6,7,8],    %DOMAIN OF X
         [1,2,3,4,5,6,7,8],    %DOMAIN OF Y
         [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],   %DOMAIN OF LEADING DIAG
         [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).  %DOMAIN OF TRAILING DIAG

queens([], [], Dy, Dl, Dt).

queens([Y|YRest], [X|DxLessX], Dy, Dl, Dt) :-   %X SELECTED IN ORDER 1 TO 8
   del(Y, Dy, DyLessY),   % CHOOSE Y
   L is X - Y,            % UPDATE DOMAINS
   T is X + Y,
   del(L, Dl, DlLessL),
   del(T, Dt, DtLessT),
   queens(YRest, DxLessX, DyLessY, DlLessL, DtLessT).
```

Another example of a resource allocation problem is the timetable problem. Suppose that you are given as input a list of modules and their weekly number of lectures such as:

```
[ 08301/2 , 08302/3 , 08303/1 , 08304/4 ]
```

The problem is to allocate these lectures in the timetable which consists of 5 possible days (Monday to Friday) and 6 possible times (9, 10, 11, 2, 3 and 4). A simple state representation would be a list of lectures to be allocated, initially the lecture domain is

```
[08301, 08301, 08302, 08302, 08302, 08303, 08304, 08304, 08304, 08304]
```

and domain of slots for a single lecture room and day is

```
[9, 10, 11, 12, 1, 2, 3, 4]
[mon, tue, wd, thu, fri]
```

and a domain of lecture rooms

```
[lta, ltb, ltc, ltd]
```

Choose a lecture from the lecture domain, reduce the lecture domain, choose a day, a lecture room and an available slot for that room.

```
alloc(mon, 9, lta, 08301)
```

Exercise: Can any of the domains be reduced at this stage? How can we ensure that the lecturer or students are not double booked? What is wrong with

```
[alloc(mon, 9, lta, 08301), alloc(mon, 9, ltb, 08301)]
```

Resource allocation is an important problem in general.

## 2.9 Priority or best first search

In both breadth-first search and depth-first search, the purpose of search is to find a goal state from the initial state as quickly as possible. Since the nodes in the queue are examined from the front, the fastest solution occurs when the goal node very quickly finds its way to the front of the queue. If this happens during Depth or Breadth first search, it will be a matter of luck rather than design. To arrange for the goal node to be examined as soon as possible, we can add the successors to the queue and then sort it with the aim of placing the best successor, i.e. the successor that will lead soonest to the goal, at the front of the queue.

This leads to a generalisation of both Depth-First and Breadth-First search that consists of arranging candidate nodes into a priority queue instead of a FIFO queue. This means that nodes are ordered according to some comparison operation, and the node with the highest priority or lowest cost is selected. In the AI literature, priority first search is known as best first search.

```
pfs(Origin, Visited) :- pfs3([Origin],    % INITIAL PRIORITY QUEUE
                              [],         % LIST OF NODES VISITED SO FAR
                              RevVisited),  % SOLUTION LIST OF NODES
                        reverse(RevVisited, Visited).

pfs3([Node|_], History, [Node | History]) :- goal(Node).

pfs3([Node|RestQ], History, RevVisited) :-
   not goal(Node),
   findall(NextNode,
           (successor(Node, NextNode),
              not member(NextNode, History),
              not member(NextNode, RestQ)),
           Successors),  %LIST OF SUCCESSORS OF Node
   addPriorityQ(RestQ, Successors, PriorityQ),    %MAKE NEW PRIORITY QUEUE
   pfs3(PriorityQ, [Node | History], RevVisited).

addPriorityQ(L1, Q1, Q2) :-  append(L1, Q1, L2),
                             sortQ(L2, Q2).

lessthan(STATE1, STATE2) :-   %STATE1 IS BETTER THAN STATE2
   ...                        %COMPLETE ACCORDING TO PROBLEM
```

After each swap, the new list is closer to a sorted list.

```
bubblesort(List, Sorted):-
      swap(List, List1), !,
      bubblesort(List1,Sorted).

bubblesort(Sorted, Sorted).  %LIST SORTED

swap([X, Y | Rest], [Y, X | Rest]):-  %SWAP FIRST PAIR
```

```
   lessthan(X, Y).

swap([Z | Rest],[Z | Rest1]):-        %SWAP PAIR IN TAIL
   swap(Rest, Rest1).
```

Insertion sort. To sort a non-empty list,

- Sort the tail.
- Insert the head into the sorted tail at such a position that the resulting list is sorted.

```
insertsort([], []).

insertsort([X|T], Sorted):-
   insertsort(T, SortedT),          %SORT THE TAIL
   insert(X, SortedT, Sorted).      %INSERT X

insert(X, [Y|Sorted], [Y|Sorted1]):-
   lessthan(X,Y), !,
   insert(X, Sorted, Sorted1).

insert(X, Sorted, [X|Sorted]).
```

## 2.10 A heuristic for the 8-Tile puzzle in Prolog

If the best order of nodes for the priority queue in known then the problem is easily solved. In general, the best order is not known but can be estimated. A heuristic is a rule or procedure which does not guarantee to provide a correct solution but is nonetheless worth using if there is nothing better because it provides an approximate solution. In Prolog, the `totalDist` heuristic has three parameters, a current state, the goal state and the total distance. It is defined as:

```
totalDist([], [], 0).

totalDist([Tile | Tiles], [Square | Squares], DistEightTiles) :-
   mandist(Tile, Square, DistOneTile),
   totalDist(Tiles, Squares, DistRestTiles),
   DistEightTiles is DistOneTile + DistRestTiles.
```

The rest of the program simply defines a `sortQ` predicate that uses the `totalDist` predicate to sort puzzles in increasing value. This is simply followed by a call to the predicate `pfs`.

The `totalDist` heuristic is not very efficient, however. We can propose a better heuristic by considering the number of tiles that are "out of sequence". An out of sequence score can be computed as follows:

- a tile in the centre counts 1; tiles can be rotated around this tile,
- a tile not in the centre counts 0 if the tile is followed by its proper successor as defined by the goal arrangement otherwise it counts 2.

and can be defined as:

```
seq([Blank, First | OtherTiles], S) :-     %S IS TOTAL SCORE
   seq([First | OtherTiles], First, S).    %KEEP FIRST TILE FOR END OF SEQ

seq([Tile1, Tile2 | Tiles], First, S) :-
   score(Tile1, Tile2, Score1),             %SCORE LEADING PAIR OF TILES
   seq([Tile2 | Tiles], First, ScoreRest),   %SCORE REMAINING PAIRS OF TILES
   S is Score1 +  ScoreRest.
```

```
seq([Last], First, S) :-     %SCORE LAST TILE FOLLOWED BY FIRST TILE
   score(Last, First, S).    %KEPT FIRST TILE TO LOOP AT END OF SEQ

score(2/2, _, 1) :- !.       % ! TO AVOID MATCH BY score(_, _, 2).

score(1/3, 2/3, 0) :- !.     % ! TO AVOID MATCH BY score(_, _, 2).
...                          %HERE ADD OTHER COORDS THAT ARE IN SEQUENCE

score(_, _, 2).              %OUT OF SEQUENCE SCORES
```

The predicate `seq` would be used sort states in the priority queue, the state with the lowest score would be at the front.

The cut, !, is used to stop prolog backtracking. It is useful when we know that a goal should be matched in only one way. We can, of course, always be sufficiently precise in how we specify the structures to be matched to ensure that a goal is matched in only one way but this might involve a lot of work. In the above example, it would mean listing all the out of sequence coordinate pairs.

Given a goal `G` which matches

```
A :- B1, B2, ... Bk, !, Bk+2, ... Bn.
```

if `G` unifies with `A` and subgoals `Bi`, i from 1 to k, succeed, i.e. the cut is reached, then the match of `G` to `A` is regarded as the only possible match for `G` and alternatives for `A` that might match `G` are ignored. Backtracking is possible for `Bi`, i greater than k, but if backtracking reaches the cut then the match of `G` to `A` fails and the search for a match proceeds from the point just before the match of `G` to `A`.

The cut is often used for efficiency.

```
min(X, Y, X) :- X <= Y, !.
min(X, Y, Y) :- X > Y, !.
```

## 2.11 Greedy algorithms or hill climbing

Clearly, the definition of `sortQ` is important but problem dependent. For some problems, the sort predicate is not difficult to define. For the majority of problems, however, it is difficult and often no perfect predicate is known. If we don't know exactly how successors should be ordered, we might, for a given problem, be able to make a reasonable estimate. A procedure which gives us such an estimate is called a heuristic. A Priority search program that uses a heuristic to sort the priority queue is called an heuristic search program.

Since the heuristic must be able to compare any two nodes in the search space, this type of heuristic is called a global heuristic. Notice that the list of nodes to test may become quite large. We thus encounter a memory and time overhead for this exhaustive sorting.

Priority-first search, combined with the need to reduce the memory and time overhead, leads us to greedy or hill-climbing algorithms. These algorithms have a much lower memory and time overhead because they maintain a much smaller priority queue.

A useful way of visualising a greedy algorithm is to imagine the search space in three dimensional terms. Up until now we have described the search space as a graph, i.e. a two dimensional structure. To extend a graph into three dimensions, consider a 2-dimensional graph in which each node represents a candidate solution. The best solution will have the lowest cost. Very poor solutions will have a very high cost. Imagine that the graph occupies the horizontal X,Y plane and that above each node, in the Z plane is a point. The height of this point above a given node is the cost of the solution represented by that node. The surface produced by these points is a three dimensional space and is the space to be searched. Bear in mind, also, that the space we have just described is typically generated on demand.

Initially we have a candidate solution with an associated cost. In terms of the 3-D surface we are some height above a solution node. The problem is to find the solution with the minimum cost. We explore (generate) the local space around the candidate. If this local space contains a solution of a lower cost than our current candidate, we greedily grab it and make it the current candidate. We have moved from one point in the 3-D surface to another of a lower height. From the new candidate we can generate more of the space to look for an even lower cost solution.

In AI terms, a greedy approach is known as “hill climbing”, although, if cost is used as an estimate then a more accurate term would be “hill descending” since each step to a lower cost solution is a step down the 3-D surface.

The crucial feature of a greedy algorithm, the thing that makes it greedy in fact, is that we commit ourselves to each step we take. We do this to save memory. We we adopt a new candidate solution with a lower cost we discard the old candidate solution. We keep doing this until we can find no step to a lower cost solution. We are in a well in the surface. This well may not, of course, be the lowest point in the surface. Since at each stage we consider only the adjacent nodes to the current node, “hill climbing” is rather like the limited vision available when descending a hill in fog. Always choosing the path which is locally down hill is no guarantee that you will reach the bottom of the hill. The search may get stuck at a local minimum. To find the optimal solution, it is necessary to climb out of the local minimum and explore new descents.

Certain problems can be solved using the greedy or hill climbing approach. These are the problems where there is a problem representation that provides a space in which there are no local minima. For some problems, it is difficult to find such a representation, for others, it is impossible. In such cases, a representation in which the local minima are almost as low and the absolute minimum may be adequate in the sense that any local minimum is a good approximate solution. In such a situation we have a heuristic algorithm. We use a heuristic to estimate the cost of a given solution. If the estimate is not too bad we may continue moving down hill in the search space but not by the most direct route.

Note that there are many state spaces for a given problem, the placing of dominoes on mutilated chess board is an example. Is is possible, using dominoes, to cover every square of a chess board from which two diagonally opposite squares have been removed.

Prolog does not have higher order predicates and so the general greedy algorithm described below is “parameterised” by the definition of `successor/2` and `sortSuccessors/2`. The candidate solution at the front of the priority queue is either the goal (solution) or we examine the successors. Only the best successor is examined.

```
greedy(Origin, Soln) :-
   greedy1([Origin],     % PRIORITY QUEUE
           [],           % NODES VISITED
           Soln).        % SOLUTION, NOT BOUND

greedy1([Node|OtherSolns], Visited, Soln) :-
   goal(Node),
   reverse([Node|Visited], Soln).

greedy1([], Visited, []).  % NO SOLUTION FOUND

greedy1([Node|OtherSolns], Visited, Soln) :-
    not (Goal = Node),
    findall(AdjNode,
            (successor(Node, AdjNode), not member(AdjNode, Visited)),
            Successors),
    sortSuccessors(Successors, PriorityQueue), !,  % ! = DON’T RESORT
    greedy1(PriorityQueue, [Node|Visited], Soln).
```

The predicate `sortSuccessors` is simply a predicate that sorts the successors according to some given heuristic. The successor at the front of the sorted list, `PriorityQueue` is the best successor and the node from which the search continues.

## 2.12 The money change problem

The problem is to express a sum of money in the smallest number of coins. Let the structure below represent the outstanding amount that remains to be changed together with a list of coins that is the changed part. For example, we might successively change 74 pence as follows

```
mcs(74, [])
mcs(24, [50])
mcs(4, [20, 50])
mcs(2, [2, 20, 50])
mcs(0, [2, 2, 20, 50])
```

Notice that at each stage, the largest possible coin is used. We can now define `goal/1` as follows

```
goal(mcs(0, _))
```

We need to define the set of available coins, we use a list.

```
coins([1, 2, 5, 10, 20, 50, 100]).

successor(mcs(M, [Change]), mcs(N, [C|Change])):-
   coins(Coins),
   member(C, Coins),
   N is M - C,
   N > -1.
```

Note that `successor/2` will try smaller coins before larger ones because of the order of the available coins in the list and the way `member/2` works. This is not ideal but is not a problem for us because the priority queue will contain all matches `successor/2` and when it is sorted, the best successor will be at the front. To sort a queue a order predicate is required. A state in which there is least outstanding money to be changed is preferred.

```
lowerCost(mcs(M, _), mcs(N, _)) :- M < N.
```

## 2.13 Converting a state space into a state path space

In assembly problems, for example, the solution, or goal state, is a sequence of assembly operations, not the finished object. In fact, there are usually many ways to assemble and object and we typically want the most efficient sequence. Each state in the problem space is a sequence of operations. An easy way of constructing this state space of sequences of operations is to work from the state space of partially assembled objects. The successor of a partially assembled object is the partially assembled object after it has undergone an assembly operation. In this space, each path corresponds to a sequence of operations. In general, given a graph, we can easily construct a new graph in which the nodes correspond to paths in the original graph.

Even though we may not be looking for a path as the solution to a problem, there is a significant advantage to searching for paths rather than nodes. Some search problems require not only that a goal is found but that the best goal is found. Searching for paths allows us to use a form of priority or best first search that will lead us to the best goal.

On the face of it, searching for paths seems a much more complicated problem than searching through nodes. In reality, it is the same problem, it is only the space that is different. If we take a space of nodes called space-A and search through paths in space-A, each path in space-A can be regarded as a node in a new space of paths, call it space-B. It is not difficult to generate the space-B of paths given the space of nodes, space-A. The successor paths of each path in the path space, space-B, can be constructed by adding the successors of the last node in space-A to the path, to generate a new extended path for each successor in space-A. This is illustrated in Figure 1.

Once we are able to generate the new path space we can search this space using any of the algorithms developed so far.

The following is a Prolog program that searches, breadth first, a graph in which each node represents a path in a related graph.

```
bfsPath(Origin, SolnPath) :- bfsPath3([[Origin]],  % LIST OF PATHS
                                      SolnPath).

bfsPath3([[Node|PathRest]|_], SolnPath) :- goal(Node),
                                            reverse([Node|PathRest], SolnPath).

bfsPath3([[Node|PathRest] | OtherPaths], SolnPath) :-
    not goal(Node),
    findall(NextNode,
            (successor(Node, NextNode),
               not member(NextNode, [Node|PathRest])),  % AVOID CIRCULAR PATH
```

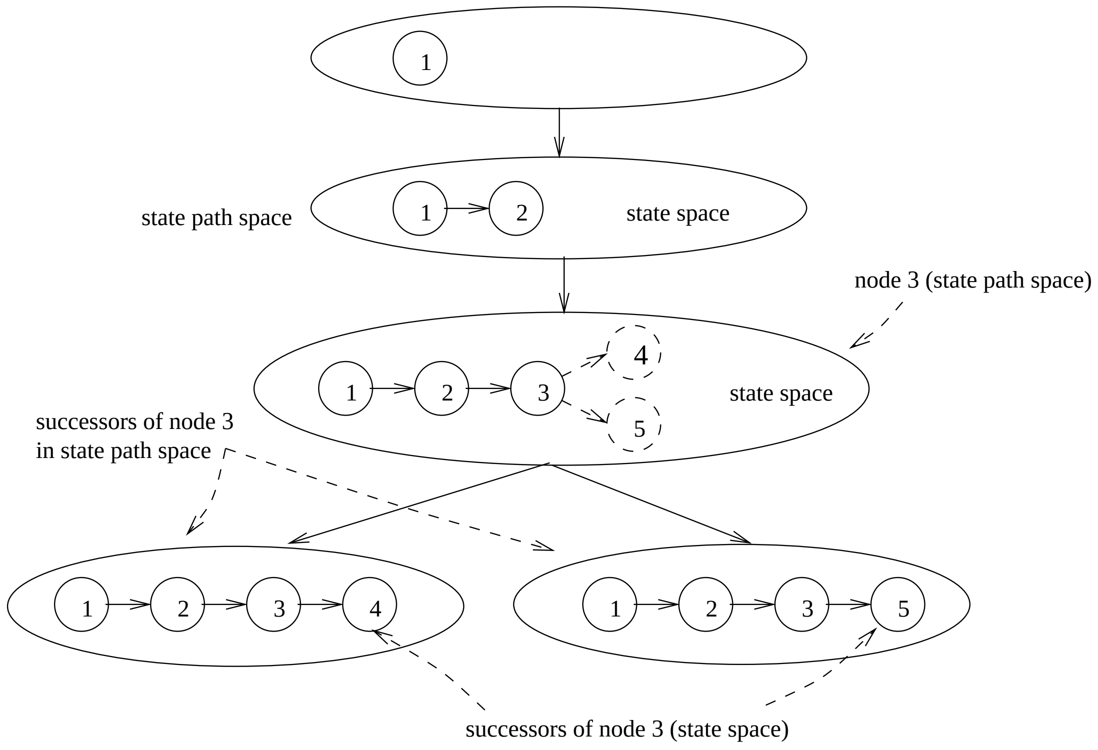

Figure 1: Transformation from state space to state path space

```
            Successors),
    extendPath(Successors, [Node|PathRest], NewPaths),
    append(OtherPaths, NewPaths, PathList),
    bfsPath3(PathList, SolnPath).

extendPath([], Path, []).

extendPath([Node | OtherNodes], Path, [[Node | Path] | OtherNodePaths]) :-
   extendPath(OtherNodes, Path, OtherNodePaths).
```

Note how we must take the first path in the queue `[Node|PathRest]` and copy it once for each successor, add each successor to the front of one copy, to generate the new paths.

## 2.14 Priority-first (best-first) path search

We have just seen how to search a space in which each node or state represents a path in some other space. Obviously, all the advantages of priority (best) first search apply in this case also, in fact the search program does not care what each node represents. We can thus use priority (best) first search to find the best path.

What we mean by best depends, of course, on the problem. It may mean the shortest path but in general, we assume a cost attached to each node-node transition. We cost a path as the sum of the arc costs, i.e. the cost of any given path is the sum of the costs of each arc in the path. This is of course a simplifying assumption that may not be true for a particular problem.

```
   Node1 --> Node2 --> Node3 --> ... Noden --> Goal
       cost12 +  cost23 + ... costnGoal = cost of path
```

If we include costs for each node-node transition then we can model many realistic search problems. The problem of finding the shortest path can be solved as a special case by finding the lowest cost path where costs are proportional to distance.

The above scheme for costing a path assumes that we have a complete path in the sense that it leads from the initial node all the way to the goal. Until the search is finished, no path is complete. How do we cost an incomplete path? We can cost only the part that we have generated.

```
start ----------------------> end of path ------------------> goal
        known path                            unknown path

 total cost = cost of known path + cost of unknown path
```

The cost of the known path can be obtained by summing the costs of all the arcs in the path, this is known. The cost of the unknown path is not known since we have not expanded that path yet. Hence we use a heuristic to estimate this cost. For each given path we can now estimate the cost of a full path to the goal which extends the given path.

Best (priority) first search using a global heuristic will therefore maintain a collection of candidate paths. For each path, we determine as best we can the final cost of that path as described above. We then select the lowest cost path and extend it by adding the arcs to the successor states of the state at the end of the path. We can extend the path to obtains paths to all the successors and these new paths are added to the queue of paths.

The priority queue ensures that the best path (as far as can be determined at this stage) is at the front. In reality, the best final path may not be at the front. This is because our estimate of the cost of the unknown part of the path may be much higher than the real cost.

```
                                        real cost 40
   start -----------------------> state-k ................... goal
               cost = 100               est cost = 20


                                        real cost 20
   start -----------------------> state-l ................... goal
               cost = 110              est cost = 100
```

The path through `state-k` is chosen in preference to that through `state-l` because it has a lower overall estimated cost of 140 as opposed to 210. In reality, the path through `state-l` is better than that through `state-k`. This means that our search program, because of the behaviour of the heuristic, may not find the best path.

There is a simple solution to this problem. We note that the problem arises because the estimate of 100 is much higher than the real cost of 20. Therefore we insist that any heuristic estimate is less than or equal to the real cost. Such a heuristic is called optimistic. In this way, we ensure that our heuristic estimate does not unfairly penalise a partial path that is a subpath of the best or optimal path.

A search algorithm is *admissible* if it always produces an optimal solution, e.g. min cost path, where a solution exists. If backtracking is used, the solution must be the first one produced. There is a famous AI algorithm called A* (A-star), very similar to the best (priority) first path search algorithm described above, which is admissible providing it uses a optimistic heuristic `h` to estimate the cost of a path from a node `n` to the goal, this is to say that `h` satisfies

```
   h(n) <= hTrue(n)   for all nodes n
```

i.e. the estimate of the cost of the path from n to the goal should be less than the true cost.

To understand why an optimistic heuristic ensures that the optimal path will be found first, imagine a situation in which a path to the goal has been found. To justifiably claim that this path is optimal, we need to be assured that there is no other lower cost path. Let us assume for the sake of argument that there is some other lower cost path. Since we did not chose this other lower cost path, its estimated cost must be higher than the cost of the path found. We know the true cost of the path found and so the estimated cost of this lower cost path must be higher than its true cost. This is to say that if there exists a lower cost path then the heuristic must have over estimated the true cost. Furthermore, if the heuristic under estimates the cost then we cannot be left in a situation in which an incomplete path has an estimated cost higher than the true cost of known path.

```
         known path => known cost
   start ----------------------> goal
```

```
|
|                   cost from node to goal estimated
|
|                       node   known cost to here
|_________________________|
         partial path
```

The true cost of the other partial path cannot be less than the known path cost if the estimate is less than the true cost.

We can trivially ensure that our best first search algorithm uses a optimistic heuristic by setting h(n) = 0 for all n. This is equivalent to using only the known portion of the path to estimate the final cost of the path. This is a very cautious but time consuming way of searching for paths and similar to breadth first search. It reduces to breadth first if all arc costs are 1.

Many kinds of optimisation are possible to the basic search program. We can avoid some of the cost of sorting by not resorting after every move to a successor. We may, for example, expand a path, add the new paths to the list and resort to find that the current path is still the cheapest. If the new paths are still the cheapest then the resorting has been a waste of time. Can optimise by keeping track of the next cheapest path in the list, i.e. the cost of the second path. When we generate a path that costs more than this second cheapest path then we resort but not otherwise.

The Prolog code for best (priority) first search of paths is easily derived from the breadth first program given earlier. The queue of paths must be sorted whenever new paths are added to it. Simply replace

```
    append(OtherPaths, NewPaths, PathList),
```

with

```
   addPriority(OtherPaths, NewPaths, PathList),
```

## 2.15 Application: cutting rectangles

A timber merchant has a need to cut rectangular pieces of wood from larger rectangles of various sizes. The required rectangles must be cut from the given rectangles in such a way as to minimise waste. We have to be careful about what it means to minimise waste. Clearly, it is more sensible to cut a rectangle out of the corner of a larger rectangle than out of the middle. In both cases, however, we have used the same area of wood.

A state of the space consists of a collection of given rectangles of wood that are available for cutting and a collection of required rectangles. In the goal state the collection of required rectangles is empty. The successor of a given state is again two collections of rectangles, there is one more given rectangle and one less required rectangle. A required rectangle can be placed on a given rectangle at the corner either horizontally or vertically. With each placing, either the horizontal or vertical cut can be extended to cut the remaining L shape into two rectangles. This means that for each selection of a given rectangle and a required rectangle in a state, there are four successors. Note that a big rectangle can not be cut from a smaller rectangle. Note also that the state space has no loops or infinite paths.

A best first heuristic search will require some estimate of the cost of a path in the space. In this application, the cost of a path is the cost of the amount of sawing or cutting done. This path cost can be determined by examining the path from the original node to the current node. We should not forget to estimate of the cost of the path remaining to the goal.

### 2.15.1 Outline of the Prolog program

The details of modelling the state are not difficult. A rectangle can be modelled as two numbers, length and breadth.

```
rectangle(3, 4).
```

Models the existence of a 3 x 4 rectangle. Since we are creating rectangles we should represent them as data, i.e. structures.

```
[rectangle(3, 4), rectangle(43, 44), ...]
```

is a list of rectangles. The predicate

```
cutRectangle(GivenRec, RequiredRec, Rect1, Rect2)
```

cuts RequiredRec from GivenRec and leaves Rect1 and Rect2.

```
successor(GivenRecList, RequiredRecList, GivenRecList1, RequiredRecList1) :-
```

chooses a rectangle from RequiredRecList and a larger rectangle from GivenRecList and produces one of the four successor states for this pair of rectangles. See `del/3` given earlier for rectangle selection.

## 2.16 Problem decomposition and and-or graphs

An and-or graph is useful when considering a problem that can be decomposed into independent subproblems. The original problem can then be solved by solving the subproblems. The subproblems must be independent, otherwise, the solution of one subproblem may undo the solution of a previous subproblem.

An example is the problem of finding a route on a map where the map contains a river which can be crossed at two bridges, bridge1 and bridge2 only. If we are looking for a route that must cross the river then we are looking for a route over bridge1 OR a route over bridge2. To find a route over bridge1 we must find a route to bridge1 AND then a route from bridge1 to the goal. We get the following and-or graph (which need not be a tree as is the case below).

```
                             path to goal
                           /              \
                          /                \
                         /                  \
          path thru bridge1                path thru bridge2
             /_________\                   /___________\
            /           \                 /             \
           /             \               /               \
path to bridge1   path from bridge1   path to bridge2   path from bridge2
```

So far we have discussed two kinds of space, a state space in which each node represents a state of the problem, and a state path space in which each node represents a path in a state space. The and-or graph is a third kind of space in which each node represents a problem to be solved.

Towers of Hanoi problem can be formulated as an and-or graph. Let `abc` be a sequence representing three discs in the order given, `a` the smallest disc and on the top. Let `abc 1-3` mean the problem of moving the discs `abc` from pole 1 to pole 3. We then have the following solution graph.

```
                        abc 1-3
                     /     |     \
                    /______|______\
                   /       |       \
                  /        |        \
               ab 1-2    c 1-3    ab 2-3
            /    |    \          /  |  \
           /_____|_____\        /___|___\
          /      |      \
         /       |       \
      a 1-3    b 1-2   a 3-2
```

Each node in the and-or graph is a problem to be solved. The leaf nodes can be solved without decomposition. In the previous discussion of state space search, the solution was a path in the graph. For an and-or graph, the solution is a subgraph of the graph.

The rules for finding the solution subgraph are:

1. Include the root or initial node
2. If the node is an or-node, include one of the successors.
3. If the node is an and-node, include all of the successors.

Exercise: Describe the execution rules for a Prolog program in terms of searching (constructing) an and-or graph. Note that Prolog does not commit itself to a successor of an or-node but backtracks to each successor as necessary.

When there is a choice of subproblems, a choice can often be made on the grounds of cost. Costs can attached to arcs or nodes. The best solution is the subgraph with the lowest total cost. The independence criterion for subproblems can be relaxed if an ordering of the subproblems can be found such that the solution of subsequent goals does not undo the solution of previous subproblems.

To search an and-or graph we can exploit prolog's inbuilt search strategy which is essentially designed to search an and-or graph. Disadvantages are that we get only a 'yes' or 'no' as a result and not the solution subgraph, cannot easily include costs, danger of infinite loop if loop in the graph.

First step to overcoming these problems is to represent the graph as a data structure rather than rely on the behaviour of the code.

```
node(A, children(or, [B, C, D]))
node(B, children(and, [E, F]))
```

Will use infix functors for readability, node becomes an arrow symbol, defined as

```
:- op(600, xfx, --->).
```

600 is the operator precedence. The highest precedence operator is the principal functor. children becomes :, defined as

```
:- op(500, xfx, :).
```

We can check any structure we build using the display predicate

```
?- display(a ---> or: [b, c, d]).
--->(a, :(or, .(b, .(c, .(d, [])))))
```

note . is the list construction functor.

The clauses below describe an and-or graph

```
a ---> or: [b, c].
b ---> and: [d, e].
c ---> and: [f, g].
e ---> or: [h].
f ---> or: [h, i].
goal(d).
goal(g).
goal(h).
```

A corresponding program to search the graph is

```
solve(Node) :- goal(Node).

solve(Node) :-
   Node ---> or: Nodes,
   member(Node1, Nodes),
   solve(Node1).

solve(Node) :-
   Node ---> and: Nodes,
   solveAll(Nodes).

solveAll([]).

solveAll([Node|Nodes]) :-
   solve(Node).
   solveAll(Nodes).
```

To avoid infinite depth of search and to produce a solution graph (cf. formation of solution path using history list of nodes visited in a graph without or-nodes), we add

```
solve(Node, Node, _) :- goal(Node).

solve(Node, Node ---> Tree, MaxDepth) :-   %MAKE Node ROOT OF TREE
   MaxDepth > 0,
   Node ---> or: Nodes,
   member(Node1, Nodes),
   Depth1 is MaxDepth - 1,
   solve(Node1, Tree, Depth1).             %Tree BOUND HERE

solve(Node, Node ---> and: Trees, MaxDepth) :-  %ADD Trees AS CHILDREN OF Node
   MaxDepth > 0,
   Node ---> and: Nodes,
   Depth1 is MaxDepth - 1,
   solveAll(Nodes, Trees, Depth1).              %Trees BOUND HERE


solveAll([], [], _).

solveAll([Node|Nodes], [Tree|Trees], _) :-
   solve(Node, Tree, _),
   solveAll(Nodes, Trees, _).
```

Note that the above program constructs a tree even if the problem graph is not a tree, common nodes are duplicated in the solution tree.

For a heuristic search we need to incorporate costs on arcs. At each node, the cost will represent the cost of solving that problem, i.e. the overall cost of the solution tree below that node. At an or-node the cost is thus the min of (cost of arc to a child plus cost at that child node)

At an and-node the cost is

sum of (cost of arc to a child plus cost at that child node)

Given these cost functions we can now use the best first search program given above to search a space of solution trees. We maintain a list of trees which we sort in the usual way according to the costs given above.

## 3 Searching Game Trees

Games such as chess and draughts are classed as two-person perfect information games. Perfect information excludes card games in which a player's hand in unknown or the cards to be dealt are unknown. A game (e.g. chess) tree is similar to a state-space graph. It is a tree because each state has a single predecessor. Two states may represent identical board positions but are not identical states because it is important to distinguish between board positions in terms of the sequence of moves that led to that position. In effect, the sequence of moves is what is really important, these paths are the real states that are being searched. Each level in the search tree of move sequences is called a ply.

Games such as chess or draughts can be formulated as and-or graphs.

```
                      white-to-move-posn-1                  ply 1
                        /            \
                       /     ...      \
                      /                \
                     /                  \
        black-to-move-posn-1      black-to-move-posn-i      ply 2
           /_________\                /___________\
          /      ...  \              /        ...  \
         /             \            /               \
        /               \          /                 \
white-to-move-posn-k                                        ply 3
```

White is able to win if there is a successor of the white-to-move-posn-1, i.e. one of black-to-move-posn-1 ... black-to-move-posn-i, which is a winning position. One of these positions is a winning position if all of white-to-move-posn-k positions are winning positions. This is because the opponent can make any of the moves that lead to one of these positions and we assume the opponent also wants to win.

The above graph can be used directly to find solution if the search space is small, in practice, the entire graph cannot be constructed. The graph is generated up to the limit of computational resources. After that, the nodes for which we have no successors must be evaluated. It is possible to evaluate the value of a board configuration without looking ahead by looking at the positions of the various pieces. If the position is an obvious win of loss then it is easy to evaluate. A win gets a very high score and a loss gets a very low score. If, as is usually the case, the position is neither an obvious win or obvious loss then a heuristic must be used. For example, in chess, pieces in the centre of the board are generally more useful than those on the edge. A queen is generally worth more than a rook. These rules are crude heuristics of course because a queen may not be worth much if she is pinned down by other pieces.

Once each successor configuration has been evaluated and given a score, the score can be used to select the next move. One player (max) will choose a move to the successor with the highest score, the opponent (min) will choose a move to the successor with the lowest score because the evaluation function does its calculations always from the point of view of the same player.

We distinguish between bottom-level values and backed-up values. Bottom-level values are obtained by consideration of the relevant board position alone. If that board position represents a win or loss for max then we can give it a very high or low score respectively. Positions other than obvious wins or losses must be assessed using some heuristic evaluation of the positions of the pieces.

A backed-up score is obtained by examining scores of positions deeper in the tree, i.e. using look-ahead. The minimax algorithm is the standard method for determining the value of a position by look ahead in a two-person game. Assumption is that the opponent is trying to win and that what is good for you is proportionately bad for the opponent, i.e. it is a zero sum game. In effect, you assume that the opponent has the same evaluation function as you but is using it in reverse.

When max makes a move, max will choose a node with the max evaluation function value. This may be a bottom-level value if at the edge of the space but usually it is a backed-up value. When it is min's turn, min will choose the node with the lowest value, assuming of course that min is trying to win.

```
                   a 1            max   BACKED-UP SCORES
                 /     \
                /       \
              b 1       c -1      min   BACKED-UP SCORES
              / \       / \
             /   \     /   \
           d 1  e 2  f -1  g 3    max   BOTTOM LEVEL SCORES
```

```
minimax(a, b, 1)
minimax(b, d, 1)
minimax(c, f, -1)

minimax(Pos, BestSucc, Val) :-
   moves(Pos, PosList), !,          %SATISFY ONCE ONLY
   best(PosList, BestSucc, Val)     %BACKED-UP SCORE
   ;                                %OR
   bottomVal(Pos, Val).             %BOTTOM LEVEL SCORE

best([Pos], Pos, Val) :-
   minimax(Pos, _, Val), !.

best([Pos1 | PosList], BestPos, BestVal) :-
   minimax(Pos1, _, Val1),
   best(PosList, Pos2, Val2),       %BEST OF THE REST
   betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterOf(Pos1, Val1, Pos2, Val2, Pos1, Val1) :-
   minToMove(Pos1),         %maxToMove AT PARENT OF Pos1
   Val1 > Val2, !
   ;
   maxToMove(Pos1),         %minToMove AT PARENT OF Pos1
   Val1 < Val2, !.

betterOf(Pos1, Val1, Pos2, Val2, Pos2, Val2).
```

The above program visits nodes in depth first order. Can improve efficiency by realising that it is not necessary to completely evaluate all nodes to find the max or min. If min is to move, we look for the min cost position. Assume the smallest cost so far found is alpha. While in the process of evaluating another of min's positions to see if it is lower than alpha, we find that the cost exceeds alpha even though we have not yet taken into account all of the successors of this position we are evaluating.

Clearly we can abandon this position and move on to min's next position with alpha still the score to beat. We keep one bound alpha for the minimum value that max can hope to achieve, and a bound beta for the maximum value max can hope to achieve. This kind of algorithm is known as alpha-beta pruning.

As an example, assume that `g` in the graph above has successors and so graph is

```
                   a 1              max
                 /     \
                /       \
              b 1       c -1        min
              / \       / \
  /   \     /   \
d 1  e 2  f -1  g 3      max
                / \
               /   \
             h 2   i 3   min
```

At `f` alpha = -1 after evaluating `h` to obtain a value of 2, it is possible to establish that the value at `g` will be at least 2 and therefore exceed alpha = -1. There is therefore no need to evaluate `i`.

Effectiveness of alpha-beta pruning depends on the order in which nodes are searched. Best to find the strong moves on each side first. This sets a small gap between bounds early on in the search and hence more pruning is done.