# B A C K W A R D  C H A I N I N G  R E A S O N I N G

Submission: You need to submit both source code and hardcopy to the lecturer during the lab class. Your hardcopy should contain source codes, test samples, and information of how to compile and run your program.

Backward chaining reasoning is one of the major reasoning mechanisms in most knowledge based systems. However, generally, backward chaining can only deal with inference rules of the form

P1^...^ Pk => P

in knowledge base (i.e. if P1 is true, ..., and Pk is true, then we infer that P is also true.

The aim of this task is to extend backward chaining function such that it can also deal with inference rules of the form

Pv...vPk => P

in knowledge base (i.e. if P1 is true, ..., or Pk is true, then we infer that P is also true. Note that here we only consider propositional case.

In particular, your knowledge base is a finite set of formulas of the following three forms:

P1^P2^...^Pk => P,
PvP2v...vPk => P, and
P,

where the third formula P is a special case of the first two formulas that their bodies are empty. Note that every Pi (i=1,...,k) and P are just propositional atoms. Then users can test if a primitive fact Q can be inferred from your knowledge base. The following is a sample execution of your program.

```
unix> mykbs
reasoning>> This is an extended propositional backward chaining
reasoning>> system. Your knowledge base can only accept
reasoning>> facts like
reasoning>> P1^P2^...^Pk=>P, or
reasoning>> P1vP2v...vPk=>P, or
reasoning>> P.
reasoning>> Now please input your knowledge base! When you finish
reasoning>> your input, just type nil!
reasoning>> AvB=>E
reasoning>> A^B=>D
reasoning>> D^E=>F
reasoning>> B^E=>F
reasoning>> A
reasoning>> B
reasoning>> C
reasoning>> nil
reasoning>> You have finished your input. Now you can test your system!
reasoning>> F?
reasoning>> yes
reasoning>> A?
reasoning>> yes
reasoning>> G?
reasoning>> no
reasoning>> D?
reasoning>> yes
reasoning>> quit
unix>
```

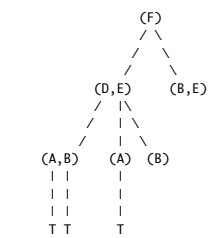
To start your implementation, you should know that backward chaining reasoning actually is a search problem. In fact, a search tree should be generated during your reasoning. However, since we require your program accept both inference rules

P1^...^Pk => P and
P1v...vPk => P,

the search tree should be considered to cooperate both conjunct and disjunct features. One of the possible structures of your search tree may be described as follows.

Suppose your knowledge base includes:
AvB=>E
A^B=>D
D^E=>F
B^E=>F
A
B
C

If user wants to test if F can be inferred from this knowledge base,
a search tree is built during the reasoning:




In the above tree, a node with the form (A,B) indicates
that both A and B should be achieved in the knowledge base
in order to achieve its parent. A node with the form (A)
indicates that A should be achieved in order to achieve its
parent. Further, to achieve a parent node, we only needs to achieve
one of its children. Therefore, a depth first search should be used
in the search procedure.

For example, to achieve F, we only need to achieve node (D,E) or
node (B,E). Then by using depth first search, we try to achieve
node (D,E). That means we need to achieve both D and E. To achieve D, we need
to achieve node (A,B), and to achieve E, we need to achieve node
(A) or node (B).

It is important to note that your program should be flexible
to accept various knowledge bases in which each fact
has one of the required forms.