**GROUP-7**

**(IIT2017005 , IIT2017010 , IIT2017021 , IIT2017023 , IIT2017028 , IIT2017030 , IIT2017036 , IIT2017503)**

**Question-1**

In this question we have to plan a route between 2 towns also giving the timetable of travel. The map data includes :-

* Mileage of vehicle
* Distance between 2 towns
* Traffic between 2 towns (higher the number , higher is the traffic)
* Road condition (higher the number , worst is the condition)
* Fuel capacity of vehicle

In this , we have described facts as edges between two towns denoted by **road(a,b,Dist,Traffic,Rc)**

where a & b are towns , Dist is distance between them , Rc is road condition between them.

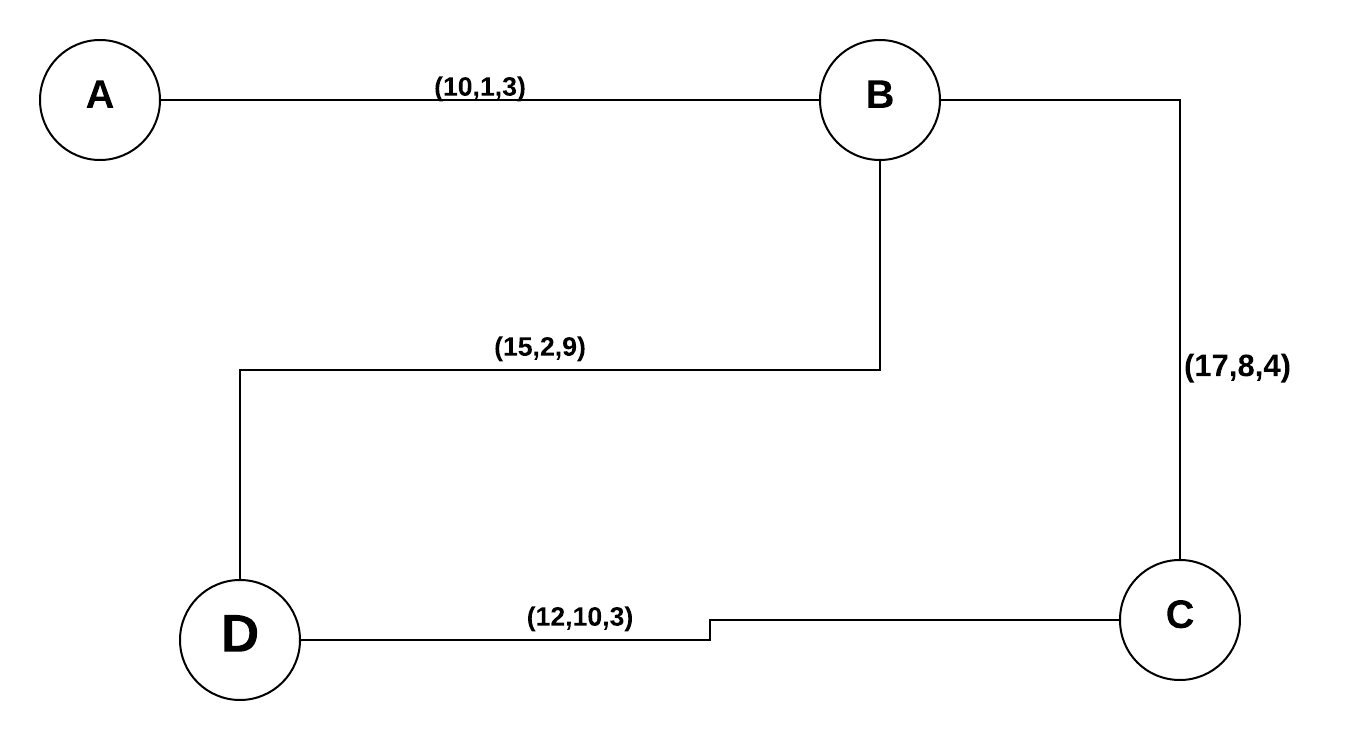
We have 3 rules :-

1. **printK(L) -** this is used to print the list L.
2. **append(X,Y,Z) -** append Y to X and store it in Z .
3. **route(A,B,L,Tlist,Cost,Fcurrent,Mil,Finitial,Wtraffic,Wroad) -** This rule works as follows:

* Find an intermediate town **X** which is connected to **A.**
* Checks if X has not been visited already using **Tlist**
* Calculates if we can move from A to X in worst case by using **Mil**(mileage of vehicle) & Distance between them , Fuel required to move from A to X. If not possible then it backtracks and find another X.
* Calculates the cost in moving till X **(Cost)** as :-
  + Cost of moving till A + Distance between A->X + (**Wtraffic** \* traffic) + (**Wroad** \* Road condition ) . Here these 2 highlited Variables are values provided by the user and are the weights which user wants to give to traffic and Road condition respectively.
* Calculates the remaining Fuel as **Frem .** if < 0 then Refuels it i.e append Refuel to the answer list **L.**
* Now recurse on finding the Route between X and B.

It terminates when we reach Destination B which is the base case . In the base case it prints the path it has taken and the total Cost associated with it.

Eg map for code -



**Question-2**

Here we have implemented a player vs. computer GO program, to find the optimal move that the computer must play so as to capture users stones. Our aim is to surround more territory than the opponent at the end of the game.

We have implemented a 5x5 Go game. We need an algorithm to capture the opponents stones so that computer tries to maximise count of its territories.

* For detecting captured areas, we run a recursive search for each position which has the opponents stones and replace it with blank position if it is captured. We find all the neighbours of each opponents stones and store it in a list and check if each position in the list is computers stones.
* If all the positions in the list for a particular enemy position is occupied by computers stone, we remove the users stone, else we check for next stone of the user.
* For predicting the position of computer’s stone we have generated a stone on the first empty position starting from first, and then run a capturing algorithm, if the prediction move given output true for capturing algorithm, this means by this predicted move we are able to capture the opponents stones, and hence is the best prediction.
* If the capturing algorithm outputs false, then for each position of the opponents stones we check how many stones we need to place to capture it, we place a stone around the one which needs minimum number of neighbours to capture.
* The board is represented by a list of 25 characters with , ‘x’,’o’ or ‘\_’ as its elements.
* The backtracking built of Prolog helps as we need to continuously check for each opponent’s position and capture if possible, else move to next position using backtracking . Prediction of moves by computer also required this recursive construct which is provided by prolog.

**Question-3**

In this Question we have to write the grammar rules to parse the sentences of some of the forms like given in the question.

In this we have used definite clause grammar rules(DCG) and defined every word in the form of nouns, verbs, noun phrase,verb phrase etc.

Rules works as below:

* The first clause defines procedure **s**, for recognizing sentences.
* An input list **S0** is passed into procedure **s**, and it must set **S** to be the remainder of the list **S** after a sentence has been removed from the beginning.
* It uses two subprocedures, it first calls **np** to remove an NP(Noun Phrase), and then it calls **VP** to remove a VP(Verb Phrase) from that. vp is followed by np.
* This decomposition continues beyond noun phrase and verb phrase until it terminates. Thus at any given step in the analysis, each part of a sentence can be seen as a **terminal** or **non-terminal**.
* Terminals would be the actual individual words (you can't analyze them further) and non-terminals would be clauses or phrases that are not yet fully broken down. So a non-terminal can be defined in terms of other elements, typically recursively, until terminals are reached.

Abbreviations are used.

"**s**" refers to "**sentence**" , "**np**" to "**noun phrase**" , "**vp**" to "**verb phrase**" , "**tv**" to "**transitive verb**" , "**n**" to "**noun**”

The code works as follows:

For example, a sentence is “**Fred saw John**”

s

np vp

det vp vp

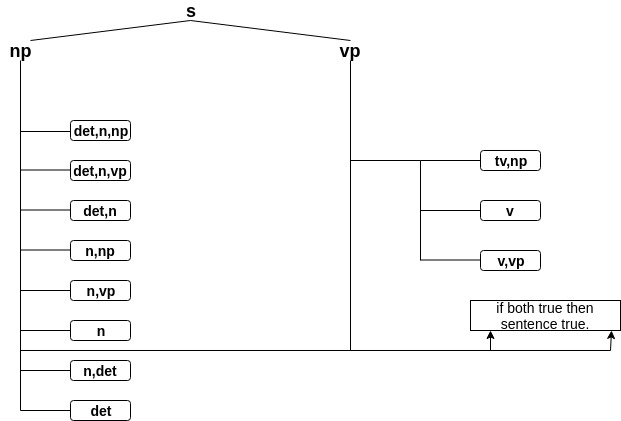
Fred tv np vp

Fred saw np vp

Fred saw det

Fred saw John

Similarly all other sentences are parsed.



**Question-4**

In this Question, we have to answer the question of the type such as how many files one own?, Did this person share this file to that person, and when the file has been changed or created by one.

For Answering these questions we are using these following facts or rules.

Rules are the following:

* **number\_of\_file(N,X):-** This rule points out the number of files X own.
* **file\_shared(F,X,Y):-** This rule points out the F file has been shared between X and Y or not.
* **question(Q,A):-** This rule points out the answer A of the desired question which has been asked(Q).

We used facts as a type of dictionary. Like

* **fileown(file, date\_created, date\_modified, owner).**
* **access\_of\_file(file, [person\_1, person\_2]).**

Strategy: Our strategy is to ask question by question(Q,A). Whatever the question has been asked is the specific format. By using the facts and rules we have retrieved the answer.

**Question-5**

In this question, we need to give game-definition relations and static evaluation function strategy of a tic-tac-toe game using alpha-beta pruning procedure.

* State of the board is shown using list where first three elements represent first row, next three elements as second row and last three elements as third row.

Some facts and rules defined for this game :

i) A player can win in this game through any column,row or diagonal. So, we have defined here :

win\_byrow(Board,Player) -> indicating player won using one of the row.

Similarly we have :

win\_bycol(Board,Player).

win\_bydiag(Board,Player).

ii) won(Board,Player). -> indicating Player has won the match and it will be true if one of - win\_byrow, win\_bycol or win\_bydiag is true.

Board is a list where if any slot is not empty, then it indicated as Player.

Three ways are there through which any player can win by row and same is for the column. A player can win by diagonal through one of the two ways and for the same , list Board will be shown accordingly:

win\_byrow(Board, Player) :- Board = [Player,Player,Player,\_,\_,\_,\_,\_,\_].

win\_byrow(Board, Player) :- Board = [\_,\_,\_,Player,Player,Player,\_,\_,\_].

win\_byrow(Board, Player) :- Board = [\_,\_,\_,\_,\_,\_,Player,Player,Player].

win\_bycol(Board, Player) :- Board = [Player,\_,\_,Player,\_,\_,Player,\_,\_].

win\_bycol(Board, Player) :- Board = [\_,Player,\_,\_,Player,\_,\_,Player,\_].

win\_bycol(Board, Player) :- Board = [\_,\_,Player,\_,\_,Player,\_,\_,Player].

win\_bydiag(Board, Player) :- Board = [Player,\_,\_,\_,Player,\_,\_,\_,Player].

win\_bydiag(Board, Player) :- Board = [\_,\_,Player,\_,Player,\_,Player,\_,\_].

iii) other(o,x) and other(x,o) -> It will help in predicting the next player to play in self play mode.

iv) game(Board,Player).

v) write([Player,Player,wins]). -> a prolog function.

vi) next\_move(Prev,Player,Newstate) -> Player plays and state of board changes as one of the slot gets filled.

vii) show\_newstate(Newstate).

viii) selfgame : When game is being played in self-mode and it will show result according to game(Board,Player).

ix) x\_wins\_in\_one(Board) -> It will be true if the Player x plays a move and wins.

x) oppo\_responses(Prev,Newstate). -> States true if state of the board changes from one state to another by one move of the opponent of x. This can happen in many cases like :

1. If o moves one step and wins.
2. If o moves one step, does not win but also x cannot win in just one move.
3. If just state of the board changes from the move of o.
4. If there is no empty slot i.e. slots already got filled.

xi) move\_ofx(list1,point,list2) -> list1 is indicating the old state and then x moves and if it possible to go to the slot ‘point’ and list2 is showing the resultant state.

xii) manual\_play. -> The 0-place predicate manual\_play starts a game with the user.

xiii) explain -> Prints what step we are going to take as player ½ and the resultant state of the board.

xiv) playingfrom(Board) -> This will work if a) x wins and it will show it using write function of prolog. b) o wins and again it will show using write function of prolog. c) One player moves, and it will be displayed and response of opponent will be there and it will be shown and recursively , playingfrom will be called for the new state of the board.

* Static Evaluation Function : When we are working for tic-tac-toe game using alpha-beta pruning, we need to measure that how good is any particular state of the board is , for a player.

We use the well known one that measures the difference between the open lines of play for each player.

Suppose we choose a player , say o and lets try to maximize its utility. So.

Larger values favor o, smaller values favor x.

* o will search to maximize a move's value. The algorithms assume that x would search to minimize it. A winning board for o has value 100, a winning board for x has value -100.
* Static Evaluation Function :

res(Board,100) :- won(Board,o), !.

res(Board,-100) :- won(Board,x), !.

res(Board,A) :- findall(o,open(Board,o),MAX) , length(MAX,Amax) , findall(x,open(Board,x),MIN) , length(MIN,Amin) , A is Amax - Amin.

Here, length(MAX,Amax) -> lines open to o

length(MIN,Amin) -> lines open to x.