University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Searching Strategies

استراتيجيات البحث

Dr. Mustafa Jaim Hadi
د. مصطفى جاسم هادي

Second Stage- First Course

cs.uotechnology.edu.iq

Problem in AI (Problem state space, Search space and Problem solving)

Search In AI:
Search is an important aspect of AI. Search can be defined as a problem-solving technique that enumerates a problem space from an initial position in search of a goal position (or solution).

The manner in which the problem space is searched is defined by the search algorithm or strategy. As search strategies offer different ways to enumerate the search space. Ideally, the search algorithm selected is one whose characteristics match that of the problem at hand.

State Space Search in AI:
The ***state space search*** is a collection of several states with appropriate connections (links) between them. Any problem can be represented as such a space search to be solved by applying some rules with technical strategy according to the suitable intelligent search algorithm.

To provide a formal description of a problem must do the following operations:

1. Define space search (state space) that contains all possible states.
2. Specify one or more states within that space that describe possible situations from which the problem-solving process may start. These states are called the initial states.
3. Specify a set of rules that describe the actions available.
4. Specify one or more states that would be acceptable as solutions ‘to the problem. These states are called goal states.
5. Determine a suitable search strategy to reach the goal.

To successfully design and implement search algorithms, a programmer must be able to analyze and predict their behavior.

Questions that need to be answered include:

• Is the problem solver guaranteed to find a solution?
• Will the problem solver always terminate, or can it become caught in an infinite loop?

• When a solution is found, is it guaranteed to be optimal?
• What is the complexity of the search process in terms of time usage? Memory usage?
• How can the interpreter most effectively reduce search complexity?
• How can an interpreter be designed to most effectively utilize a representation language?

A first part presents a set of definitions and concepts that lay the foundations for the search procedure into which induction is mapped. The second part presents an alternative approach that has been taken to induction as a search procedure and finally the third part presents the version space as a general methodology to implement induction as a search procedure.

If the search procedure contains the principles of the above three requirement parts, then the search algorithm can give a guarantee to get an optimal solution for the current problem.

Some common terms in the searching issues

State:
State is a representation that an agent can find itself in.

State Space:
A state space is a graph whose nodes are the set of all states, and whose links are actions that take the agent from one state into another.

Search Tree:
A search tree is a tree in which the root node is the start state and has a reachable set of children.

Search Node:
A search node is a node in the search tree.

Goal:
A goal is a state that the agent is trying to reach.

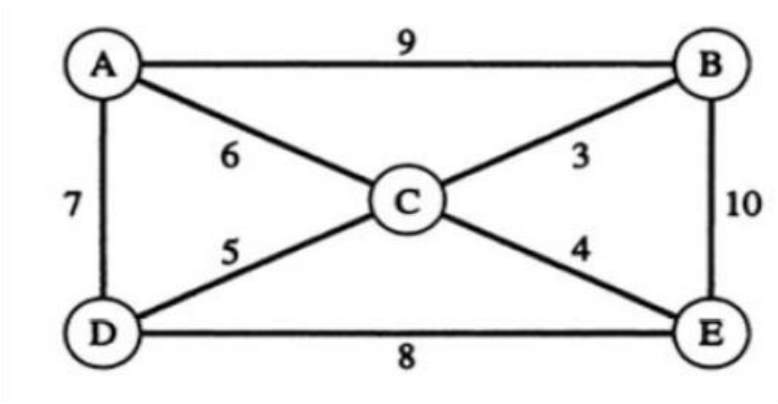
Action:
An action is something that the agent can choose to do.

Branching Factor:
The branching factor in a search tree is the number of actions available to the

agent.

Example: Describe and give an example for the Travelling Salesman Problem (TSP) as a state space?

Solution:
Given an undirected weighted graph, we should find a shortest tour (a shortest path in which every node (city) is visited exactly once, except that the initial and terminal nodes are the same).
Figure below shows an example of such a graph and its optimal solution. A, B, etc., are cities and the numbers associated with the links are the distances between the cities.



Optimal solution:

A —9— B —3— C —4— E —8— D —7— (A)

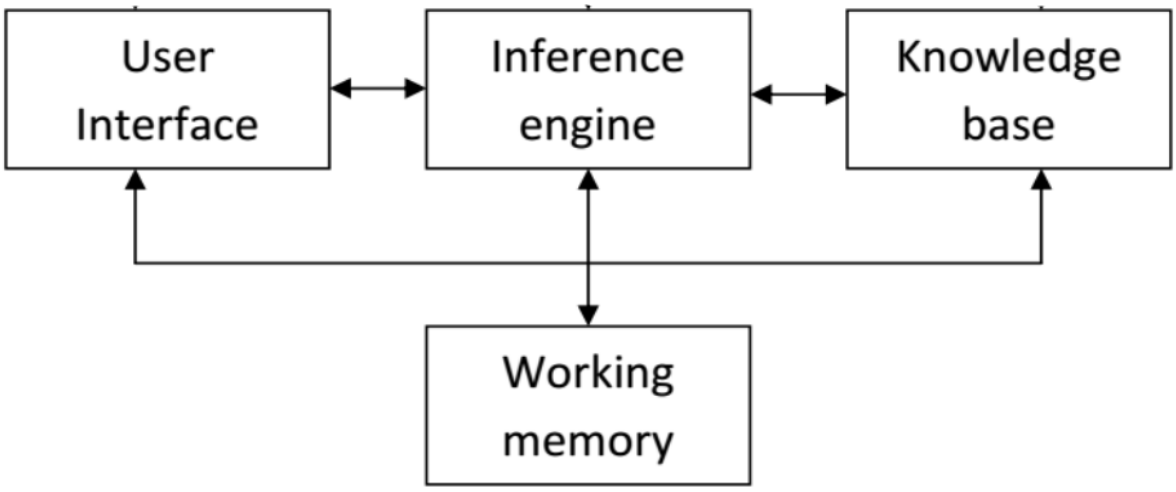
AI production system components (Knowledge base, Inference engine, Working memory and User Interface)

AI production system components:

AI production system (also called a rule-based system) is a system based on IF… THEN… rules and consisting of following components:

1. The knowledge base (production rules): is condition-action pair presented in the following form "IF condition THEN action", and it represents as a single chunk of problem-solving knowledge. The condition part of the rule is a pattern that determines when the rule may be applied to a problem. The action part defines the associated problem-solving step.
2. The inference engine (control structure): It implements search allowing the AI system to move towards a goal within the set of rules. The inference engine also called a control structure, an interpreter or a recognize-act cycle.
3.The working memory: It contains a description of the current state of the problem-solving.
4. User Interface: A user interface is the method by which the AI production system interacts with a user. These can be through dialog boxes, command prompts, forms, or other input methods. Some AI systems interact with other computer applications, and do not interact directly with a human. In these cases, the AI system will have an interaction mechanism for transactions with the other application, and will not have a user interface.

The following diagram show the AI production system components:



AI production system types:

There are three broad kinds of production system: forward, backward chaining, and rule cycle systems.
In a forward chaining system, you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts.
In a backward chaining system, you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new subgoals to prove as you go.
Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven.
The Rule Cycle (Hybrid method) refers to the repetitive process by which a rule-based system evaluates, selects, and applies rules until a specified goal is achieved or no further rules can be applied.

1. Forward Chaining

Definition:
Forward chaining is a data-driven approach used in rule-based systems where the inference engine starts with the available data and applies rules to extract more data until a goal is reached.

Process:
Start with Initial Facts: Begin with a set of known facts.
Apply Rules: Evaluate which rules can be applied based on these facts.
Infer New Facts: Apply the rules to infer new facts.
Repeat: Continue applying rules and inferring new facts until the desired goal or conclusion is reached, or no more rules can be applied.

***Example*:**
In a medical diagnosis system:

Initial fact: The patient has a fever.
Rule 1: IF a patient has a fever AND a cough THEN the patient may have the flu.
Rule 2: IF a patient may have the flu THEN suggest a flu test.
The system infers that since the patient has a fever and cough, they may have the flu and therefore suggests a flu test.

2. Backward Chaining

Definition:
Backward chaining is a goal-driven approach where the inference engine starts with the goal and works backward to determine which facts and rules support the goal.

Process:
Start with Goal: Begin with a specific goal or hypothesis.
Find Supporting Rules: Determine which rules could lead to the goal.
Check Facts: Verify if the conditions of these rules are satisfied by existing facts.
Work Backwards: Continue working backwards, checking conditions and sub-goals, until the initial facts support the goal, or no more rules apply.

***Example*:**
In a troubleshooting system:

Goal: Determine if the network issue is due to a disconnected cable.
Rule: IF the network is down AND the cable is disconnected THEN the issue is a disconnected cable.
Fact: The network is down.
The system works backward to check if the cable is disconnected, leading to the conclusion about the network issue.

3. Rule Cycle Systems

Definition:
A rule cycle system is a rule-based system that continuously evaluates and applies rules in a cyclic manner until a goal is achieved or no further rules can be applied.

Process:
Match: Identify which rules are applicable based on current facts.
Conflict Resolution: Select one rule to apply from the set of applicable rules.
Execute: Apply the selected rule, updating the system's working memory.
Repeat or Terminate: Check if the goal is achieved or if further rules can be applied. Repeat the cycle if necessary.

***Example*:**
In a home automation system:

Initial fact: The room temperature is high.
Rule 1: IF the room temperature is high THEN turn on the air conditioner.
Rule 2: IF the air conditioner is on AND the room temperature drops THEN turn off the air conditioner.
The system cycles through matching, selecting, and applying rules to maintain the desired room temperature.

Forward Chaining System:

In a forward chaining system, the facts in the system are represented in a working memory which is continually updated. Rules in the system represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called condition-action rules. The conditions are usually patterns that must match items in the working memory, while the actions usually involve adding or deleting items from the working memory.

The control structure will control the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a recognize-act cycle. The system first checks to find all the rules whose conditions hold, given the current state of working memory. It then selects one and performs the actions in the action part of the rule. The selection of a rule to fire is based on fixed strategies, known as conflict resolution strategies.

The actions will result in a new working memory, and the cycle begins again. This cycle will be repeated until either no rules fire, or some specified goal state is satisfied. Rule-based systems vary greatly in their details and syntax, so the following examples are only illustrative.

First we'll look at a very simple set of rules:
1. IF lecturing(X) AND marking-practicals(X) THEN ADD (overworked(X))
2. IF month(february) THEN ADD (lecturing(john))
3. IF month(february) THEN ADD (marking-practicals(john))
4. IF overworked(X) OR slept-badly(X) THEN ADD (bad-mood(X))

5. IF bad-mood(X) THEN DELET (happy(X))
6. IF lecturing(X) THEN DELET (researching(X))

Let us assume that initially we have a working memory with the following facts:
month(february)
happy(john)
researching(john)

Production system will first go through all the rules checking which ones apply given the current working memory. Rules 2 and 3 both apply, so the system has to choose between them, using its conflict resolution strategies. Let us say that rule 2 is chosen. So, lecturing(john) is added to the working memory, which is now:

lecturing(john)
month(february)
happy(john)
researching(john)

Now the cycle begins again. This time rule 3 and rule 6 have their preconditions satisfied. Lets say rule 3 is chosen and fires, so marking-practicals(john) is added to the working memory.

On the third cycle rule 1 fires, so, with X bound to john, overworked (john) is added to working memory which is now:

overworked(john)
marking-practicals(john)
lecturing(john)
month(february)
happy(john)
researching(john)

Now rules 4 and 6 can apply. Suppose rule 4 fires, and bad-mood(john) is added to the working memory.
And in the next cycle rule 5 is chosen and fires, with happy(john) removed from the working memory.
Finally, rule 6 will fire, and researching(john) will be removed from working memory, to leave:

bad-mood(john)
overworked(john)
marking-practicals(john)
lecturing(john)
month(february)

The five facts in the working memory imply that there is a person called "john" that works as lecturing and marking-practical at the same time in the February month and this cause an overworked load and thus he has a bad-mood state.

Forward Chaining Algorithm (Data-driven search algorithm)

1. Begins with a pattern (a problem description) added to the working memory.
2. The control structure compares matching of the pattern with IF part of rules in the production rules.
3. Firing a rule, its THEN part is added to the working memory and the process continues.
4. Search stops when the goal is found.

Example1: Suppose you have the following production rules:
1. IF John is a student THEN John enjoys student's life
2. IF John enjoys student's life
   THEN John meets friends AND John participates in university's events
3. IF John meets friends THEN John needs money
4. IF John needs money THEN John has a job
5. IF John meets friends AND John participates in university's events
   THEN John has little free time
6. IF John has little free time AND John has a job
   THEN John is not successful in studies
      AND John does not receive scholarship

Trace the Forward Chaining Algorithm using the given production rules to get the goal "John does not receive scholarship" from the start sentence that is "John is a student". Show the contents of the Working Memory and the Conflict Set (i.e., all the rules that match the facts in the Working Memory) and the Rule Fired (i.e., select one member of conflict set to execute).

Note: You can rewrite all the sentences to atomic sentences (for example; John is a student become as john_is_a_student) or you can only write them as letters (for example; John is a student will be **A** and John enjoys student's life is **B** and so on).

Solution:
At the first, we can assume that each sentence just as a letter, so the production rules will become as follows:

1. **IF** John is a student **THEN** John enjoys student's life
2. **IF** John enjoys student's life **THEN** John meets friends **AND** John participates in university's events
3. **IF** John meets friends **THEN** John needs money
4. **IF** John needs money **THEN** John has a job
5. **IF** John meets friends **AND** John participates in university's events **THEN** John has little free time
6. **IF** John has little free time **AND** John has a job **THEN** John is not successful in studies **AND** John does not receive scholarship

1. **IF** A **THEN** B
2. **IF** B **THEN** C **AND** D
3. **IF** C **THEN** E
4. **IF** E **THEN** F
5. **IF** C **AND** D **THEN** G
6. **IF** G **AND** F **THEN** H **AND** I

Start - A
Goal - I

The tracing for this algorithm using the given production rules will be shown as follows:

Iteration	Working memory	Conflict set	Fired rule
0	A	1	1
1	A, B	1, 2	2
2	A, B, C, D	1, 2, 3, 5	5
3	A, B, C, D, G	1, 2, 3, 5	3
4	A, B, C, D, G, E	1, 2, 3, 5, 4	4
5	A, B, C, D, G, E, F	1, 2, 3, 5, 4, 6	6
6	A, B, C, D, G, E, F, H, I	1, 2, 3, 5, 4, 6	goal

Then there are 8 facts which must to be found to reach to the goal I="John does not receive scholarship" and are:

1- A=John is a student.
2- B=John enjoys student's life.
3- C=John meets friends.
4- D=John participates in university's events.
5- G=John has little free time.
6- E=John needs money.
7- F=John has a job.
8- H=John is not successful in studies.

The production rules that are used to get these facts are {1,2,3,5,4,6}.

Example2: Suppose you have the following production rules:

1. **IF** p **AND** q **THEN** goal
2. **IF** r **AND** s **THEN** p
3. **IF** w **AND** r **THEN** p
4. **IF** t **AND** u **THEN** q
5. **IF** v **THEN** s
6. **IF** start **THEN** v **AND** r **AND** q

Write the Forward Chaining Algorithm and then trace this algorithm using the given production rules to verify the goal through the start fact. Draw AND/OR graph for the tracing of this algorithm.

Solution:

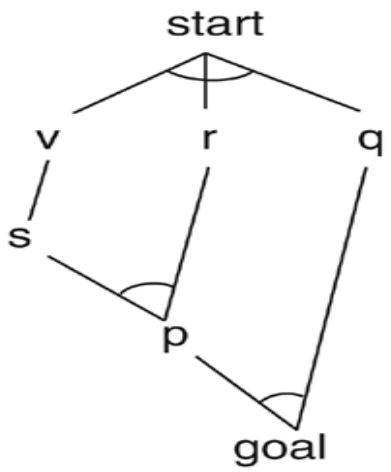
Forward Chaining Algorithm (Data-driven search algo.):

1. Begins with a pattern (a problem description) added to the working memory.
2. The control structure compares matching of the pattern with IF part of rules in the production rules.
3. Firing a rule, its THEN part is added to the working memory and the process continues.
4. Search stops when the goal is found.

The tracing for this algorithm using the given production rules will be shown as follows:

Iteration #	Working memory	Conflict set	Rule fired
0	start	6	6
1	start, v, r, q	6, 5	5
2	start, v, r, q, s	6, 5, 2	2
3	start, v, r, q, s, p	6, 5, 2, 1	1
4	start, v, r, q, s, p, goal	6, 5, 2, 1	halt

The AND/OR graph for this tracing is shown as follows:



Example3: Trace the Forward Chaining Algorithm using the following production rules to sorting a string cbaca to aabcc.
Production rules:

1. **IF ba THEN ab**
2. **IF ca THEN ac**
3. **IF cb THEN bc**

Solution:

Production rules:
1. IF ba THEN ab
2. IF ca THEN ac
3. IF cb THEN bc
START= cbaca GOAL= aabcc.

Iteration	Working Memory	Conflict Set	Rule Fired
0	**cbaca**	1,2,3	1
1	cbaca,c**ab**ca	2	2
2	cbaca,cabca,**ac**bca	2,3	2
3	cbaca,cabca,acbca,acb**ac**	1,3	1
4	cbaca,cabca,acbca,acbac,ac**ab**c	2	2
5	cbaca,cabca,acbca,acbac,acabc,a**ac**bc	3	3
6	cbaca,cabca,acbca,acbac,acabc,aacbc,aa**bc**c	-	Halt

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

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

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

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

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

test3(m,c,s):-
     confirm(has,stripes),
     asserta(have_found(tiger)),!.
test3(m,c,n):-
     asserta(have_found(cheetah)).
test3(m,c,l):-
     not(confirm(does,swim)),not(confirm(does,fly)),!.
```

```
test3(m,n,n):-
      asserta(have_found(blue_whale)).
test3(n,n,f):-
      confirm(does,fly),asserta(have_found(eagle)),!.
%test3(n,n,n):-
      %asserta(have_found(ostrich)).
test3(n,w,t):-
      confirm(has,tentacles),asserta(have_found(octopus)),!.
test3(n,w,n).

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

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

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

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

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

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

Backward Chaining System

So far, we have looked at how rule-based systems can be used to draw new conclusions from existing data, adding these conclusions to a working memory. This approach is most useful when you know all the initial facts, but don't have much idea what the conclusion might be.

If you do know what the conclusion might be, or have some specific hypothesis to test, forward chaining systems may be inefficient. You could keep on forward chaining until no more rules apply or you have added your hypothesis to the working memory. But in the process the system is likely to do a lot of irrelevant work, adding uninteresting conclusions to working memory. For example, suppose we are interested in whether **john** is in a **bad-mood**.

We could repeatedly fire rules, updating the working memory, checking each time whether (**bad-mood john**) is in the new working memory. But maybe we had a whole batch of rules for drawing conclusions about what happens when I'm **lecturing**, or what happens in **February** - we really don't care about this, so would rather only have to draw the conclusions that are relevant to the goal.

This can be done by backward chaining from the goal state (or on some hypothesized state that we are interested in).
This is essentially what Prolog does, so it should be fairly familiar to you by now. Given a goal state to try and prove (e.g., *bad-mood(john)*) the system will first check to see if the goal matches the initial facts given. If it does, then that goal succeeds. If it doesn't the system will look for rules whose conclusions (previously referred to as actions) match the goal.

One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting these as new goals to prove. Note that a backward chaining system does NOT need to update a working memory. Instead, it needs to keep track of what goals it needs to prove to prove its main hypothesis.

In principle we can use the same set of rules for both forward and backward chaining. However, in practice we may choose to write the rules slightly differently if we are going to be using them for backward chaining. In backward chaining we are concerned with matching the conclusion of a rule against some goal that we are trying to prove. So the 'then' part of the rule is usually not expressed as an action to take (e.g., add/delete), but as a state

which will be true if the premises are true.

So, suppose we have the following rules:

1. IF *lecturing(X)* AND *marking-practicals(X)* THEN *overworked(X)*
2. IF *month(february)* THEN *lecturing(john)*
3. IF *month(february)* THEN *marking-practicals(john)*
4. IF *overworked(X)* THEN *bad-mood(X)*
5. IF *slept-badly(X)* THEN *bad-mood(X)*

And there is only one initial fact in the worked memory is: *month(february),* and we're trying to prove *bad-mood(john)*

First we check whether the goal state (*bad-mood(john)*) is in the initial facts in the working memory. As it isn't there, we will add the goal to working memory and then we try matching it against the conclusions of the rules in the Production Rules.

It matches **rules 4** and **5**. Let us assume that **rule 4** is chosen first - it will try to prove *overworked(john)*. **Rule 1** can be used, and the system will try to prove *lecturing(john)* and *marking-practicals(john)*. Trying to prove the first goal, it will match **rule 2** and try to prove *month(february).* This is a fact in the working memory. We still have to prove *marking-practicals(john)*. **Rule 3** can be used, try to prove *month(february)*. This is a fact in the working memory and in this place we have proved the original goal *bad-mood(john).*

One way of implementing this basic mechanism is to use a stack of goals still to satisfy. You should repeatedly pop a goal of the stack, and try and prove it. If its in the set of initial facts then its proved. If it matches a rule which has a set of preconditions then the goals in the precondition are pushed onto the stack. Of course, this doesn't tell us what to do when there are several rules which may be used to prove a goal.

If we were using Prolog to implement this kind of algorithm we might rely on its backtracking mechanism - it'll try one rule, and if that results in failure it will go back and try the other. However, if we use a programming language without a built in search procedure we need to decide explicitly what to do. One good approach is to use an agenda, where each item on the agenda represents one alternative path in the search for a solution.

The system should try expanding each item on the agenda, systematically trying all possibilities until it finds a solution (or fails to). The particular method used for selecting items off the agenda determines the

search strategy - in other words, determines how you decide on which options to try, in what order, when solving your problem. We'll go into this in much more detail in the section on search.

Backward Chaining Algorithm (Goal-driven search algorithm)

1. A goal (a pattern) is added to the working memory.
2. The control structure compares matching of the pattern with THEN part of rules in the production rules.
3. Firing a rule, its IF part is added to the working memory and the process continues.
4. Search stops when facts on problem are found.

Example: Suppose you have the following production rules:

1. **IF** p **AND** q **THEN** goal
2. **IF** r **AND** s **THEN** p
3. **IF** w **AND** r **THEN** p
4. **IF** t **AND** u **THEN** q
5. **IF** v **THEN** s
6. **IF** start **THEN** v **AND** r **AND** q

Write the Backward Chaining Algorithm and then trace this algorithm using the given production rules to verify the start fact through the goal. Draw AND/OR graph for the tracing of this algorithm.

Solution:

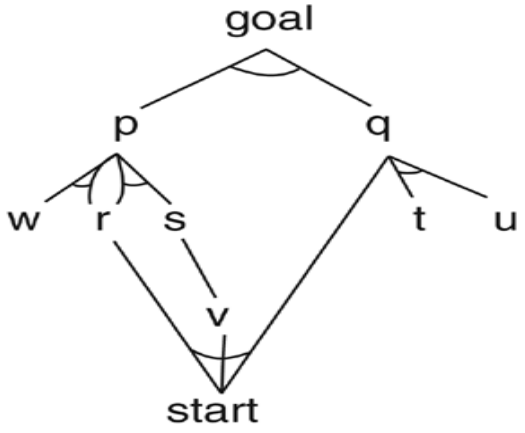
Backward Chaining Algorithm (Goal-driven search algo.):

1. A goal (a pattern) is added to the working memory.
2. The control structure compares matching of the pattern with THEN part of rules in the production rules.
3. Firing a rule, its IF part is added to the working memory and the process continues.
4. Search stops when facts on problem are found.

The tracing for this algorithm using the given production rules will be shown as follows:

Iteration #	Working memory	Conflict set	Rule fired
0	goal	1	1
1	goal, p, q	1, 2, 3, 4	2
2	goal, p, q, r, s	1, 2, 3, 4, 5	3
3	goal, p, q, r, s, w	1, 2, 3, 4, 5	4
4	goal, p, q, r, s, w, t, u	1, 2, 3, 4, 5	5
5	goal, p, q, r, s, w, t, u, v	1, 2, 3, 4, 5, 6	6
6	goal, p, q, r, s, w, t, u, v, start	1, 2, 3, 4, 5, 6	halt

The AND/OR graph for this tracing is shown as follows:



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

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

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

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

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

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

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

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

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

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

n_confirm:-
```

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

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

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

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

goal:guess_animal.

/*
has it hair
no
does it give_milk
no
has it feathers
yes
does it lay_egges
yes
does it fly
yes
has it pointed_teeth
no
does it eat_meat
yes
has it use_as_national_symbol
yes
Your animal is a(n)eagle

DataBase for correct replies are:
db_confirm("has","feathers")
```

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

Determining the control strategy for some problems

Whether you use forward or backward chaining to solve a problem depends on the properties of your rule set and initial facts. So, we must understand the comparison between forward or backward reasoning.

Forwards vs. Backwards Reasoning

Forward chaining is the best choice if:

1- All the facts are provided with the problem statement; or
2- There are many possible goals, and a smaller number of patterns of data; or
3- There isn't any sensible way to guess what the goal is at the beginning of the consultation.

Backward chaining is the best choice if:

1- The goal is given in the problem statement; or
2- The system has been built so that it asks for pieces of data rather than expecting all the facts to be presented to it.

Path Building Using Forward Chaining

Forward chaining is a method used in rule-based systems to derive conclusions from a given set of facts by repeatedly applying inference rules. It is data-driven and starts with the initial facts, applying rules to generate new facts until a specific goal is reached. This method is especially useful for pathfinding problems, such as navigating a maze or finding routes in a network.

Detailed Process of Forward Chaining for Path Building

1. Initialization

Start with a set of known facts about the environment (e.g., connections between nodes in a graph).

Define the goal state that needs to be reached.

2. Inference Rules

Define rules that describe how new facts can be inferred from existing facts. In the context of pathfinding, these rules will typically describe how to move from one node to another if certain conditions are met.

3. Iterative Application of Rules

Continuously apply the rules to the known facts to infer new facts.

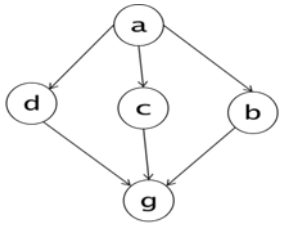
Update the set of known facts with each iteration.

4. Termination

The process terminates when the goal state is inferred (i.e., when a path to the goal is found) or when no more new facts can be inferred.

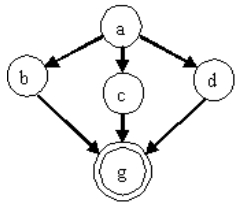
Example: There are three routes in the shape below. We can use forward chaining for building a path form a to g. There are three main movements. The movements are from (a to d and g), (a to c and g), and (a to b and g). Each moving is consisted of two movements:

Moving from a t o d
Moving from d t o g
Moving from a t o c
Moving from c t o g
Moving from a t o b
Moving from b t o g



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

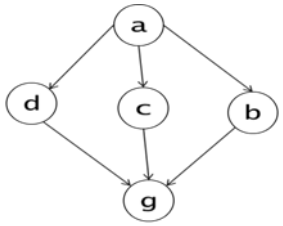
```
predicates
   run(char,char).
   find_rout (char,char).
   path(char,char).
   write_rout.
database
   rout(char,char).
clauses
   run(_,_):-retractall(_),fail.
   run(S,E):-find_rout(S,E),fail.
   run(_,_):-write_rout.

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

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

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



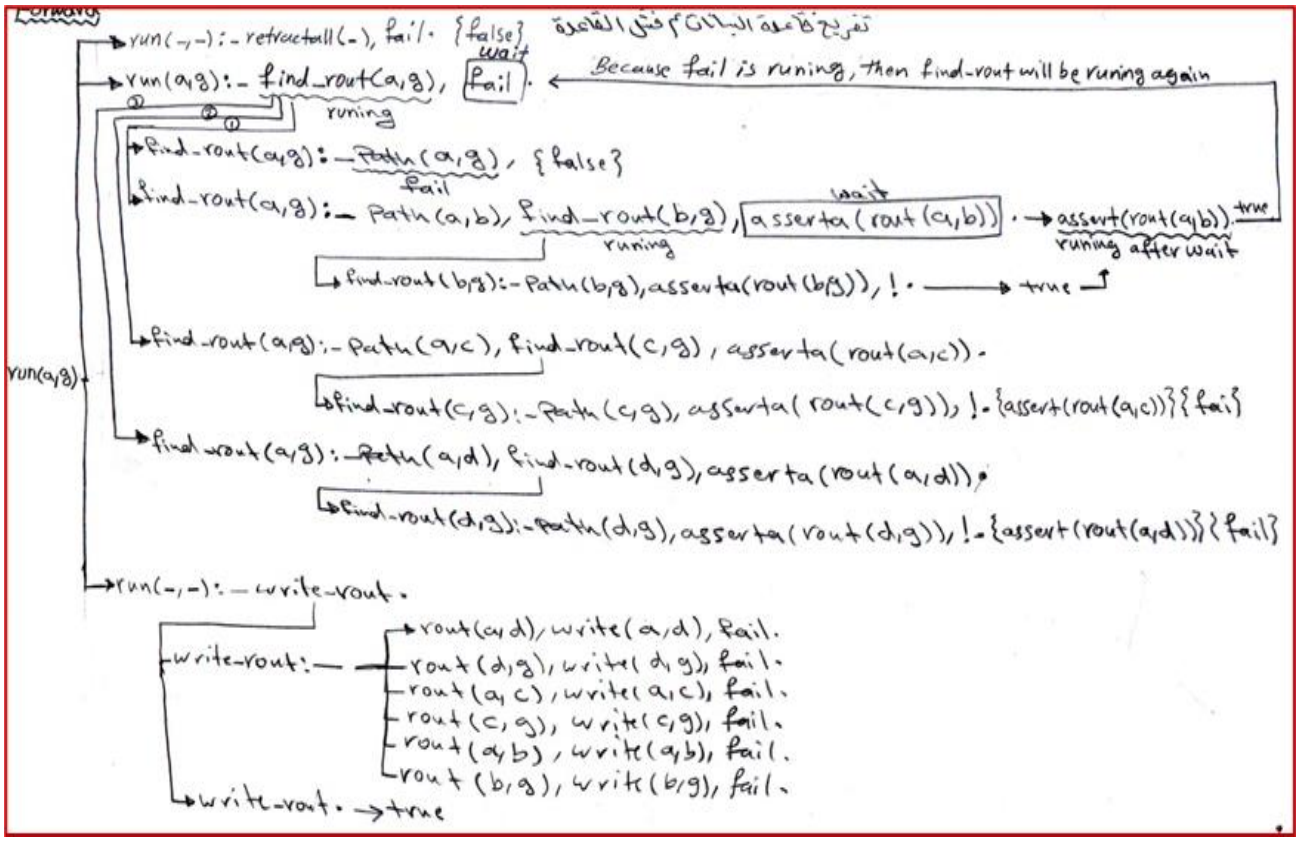

```
/*goal:
          run('a','g').
   Searching from    a   t o   d
   Searching from    d   t o   g
   Searching from    a   t o   c
   Searching from    c   t o   g
   Searching from    a   t o   b
   Searching from    b   t o   g
   yes*/
```

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

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



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



Path Building Using Backward Chaining

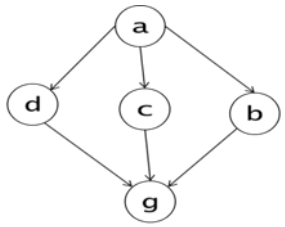
Backward chaining is a method used to find paths or achieve goals by starting from the goal and working backward to identify the necessary conditions.

Detailed Process of Backward Chaining for Path Building:

1. **Define the Goal**: Clearly state the desired outcome (e.g., reaching a location).
2. **Identify Rules**: List the conditions or rules that lead to the goal.
3. **Start with the Goal**: Ask what must be true to achieve this goal.
4. **Work Backward**: For each condition, determine the previous states or actions needed to satisfy those conditions.
5. **Construct the Path**: Build a sequence of actions from the starting point to the goal based on the identified conditions.
6. **Verify the Path**: Ensure that each step logically leads to the next and satisfies the goal.

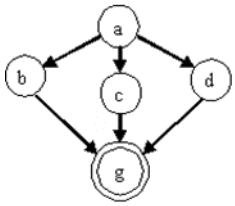
Example: There are three routes in the shape below. We can use backward chaining for building a path form a to g. There are three main movements. The movements are from (g to d and a), (g to c and a), and (g to b and a). Each moving is consisted of two movements:

Searching from g to d
Searching from d to a
Searching from g to c
Searching from c to a
Searching from g to b
Searching from b to a



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

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

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

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

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

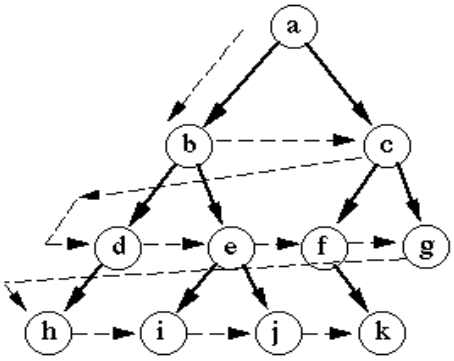



```
/*goal:
          run('a','g').
     Searching from    g   t o   d
     Searching from    d   t o   a
     Searching from    g   t o   c
     Searching from    c   t o   a
     Searching from    g   t o   b
     Searching from    b   t o   a
    Yes */
```

Uninformed Search (Blind Search)

1-Breadth – First – Search

In breadth –first search, when a state is examined, all of its siblings are examined before any of its children. The space is searched level-by-level, proceeding all the way across one level before doing down to the next level.

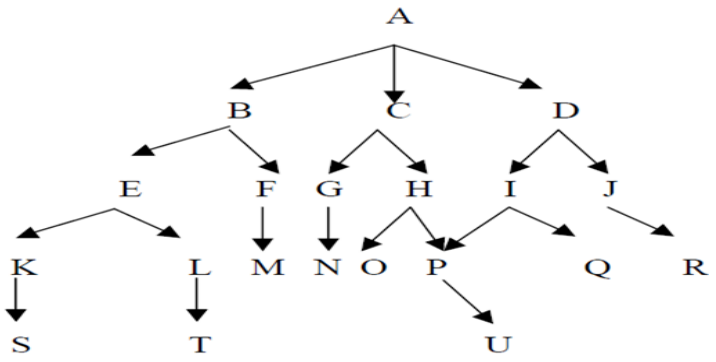


Breadth-first search

Breadth – first – search Algorithm

Begin
Open: = [start];
Closed: = [ ];
While open ≠ [ ] do
Begin
Remove left most state from open, call it x;
If x is a goal the return (success)
Else
Begin
Generate children of x;
Put x on closed;
Eliminate children of x on open or closed;
Put remaining children on right end of open
End
End
Return (failure)
End.

Example:



Breadth – first – search

1 – Open = [A]; closed = [ ].

2 – Open = [B, C, D]; closed = [A].

3 – Open = [C, D, E, F]; closed = [B, A].

4 – Open = [D, E, F, G, H]; closed = [C, B, A].

5 – Open = [E, F, G, H, I, J]; closed = [D, C, B, A].

6 – Open = [F, G, H, I, J, K, L]; closed = [E, D, C, B, A].

7 – Open = [G, H, I, J, K, L, M]; closed = [F, E, D, C, B, A].

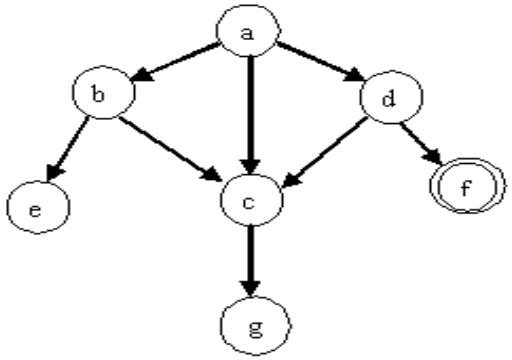
8 – Open = [H, I, J, K, L, M, N,]; closed = [G, F, E, D, C, B, A].

9 – and so on until either U is found or open = [ ].

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

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

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

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

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

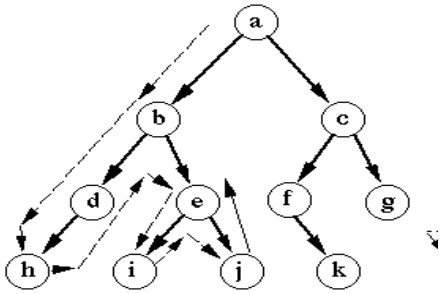
```
        append(T,L,M).

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

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




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

2-Depth – first – search

In depth – first – search, when a state is examined, all of its children and their descendants are examined before any of its siblings. Depth – first search goes deeper in to the search space whenever this is possible only when no further descendants of a state cam found owe its



Depth-first search

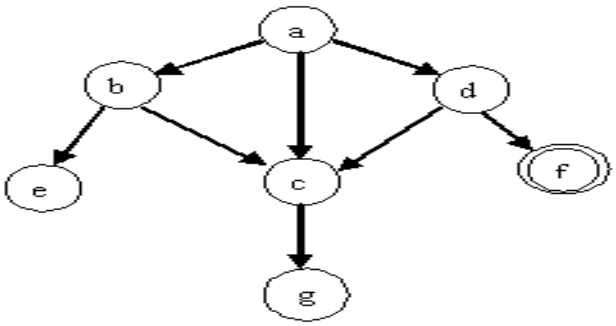
Depth – first – search Algorithm

Begin
Open: = [start];
Closed: = [ ];
While open ≠ [ ] do
Remove leftmost state from open, call it x;
If x is a goal then return (success)
Else begin
Generate children of x;
Put x on closed;
Eliminate children of x on open or closed; put remaining children on left end of open end
End;
Return (failure)
End.

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

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

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

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

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

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

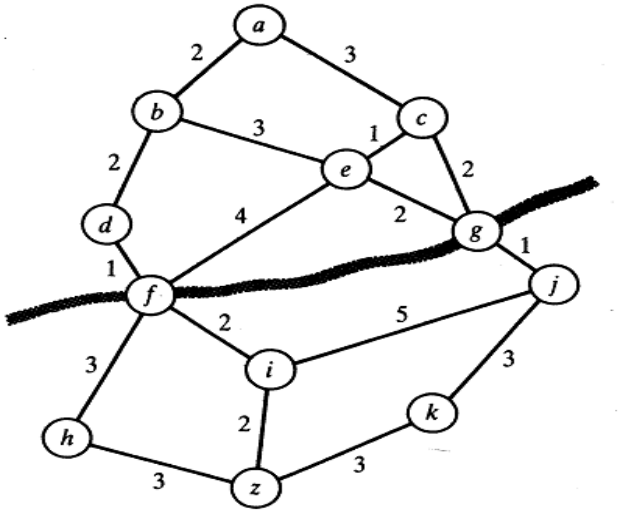
```
        append(T,L,M).

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

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




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

Problem Reduction Using AND/OR Graphs

AND/OR graphs are a suitable representation for problems that can be naturally decomposed into mutually independent subproblems. Examples of such problems include route finding, symbolic integration, game playing, theorem proving, etc. The AND/OR graph representation relies on the decomposition of problems into subproblems. Decomposition into subproblems is advantageous if the subproblems are mutually independent, and can therefore be solved independently of each other. Let us illustrate this with an example. Consider the problem of finding a route in a road map between two given cities, as shown in Figure (1) below:



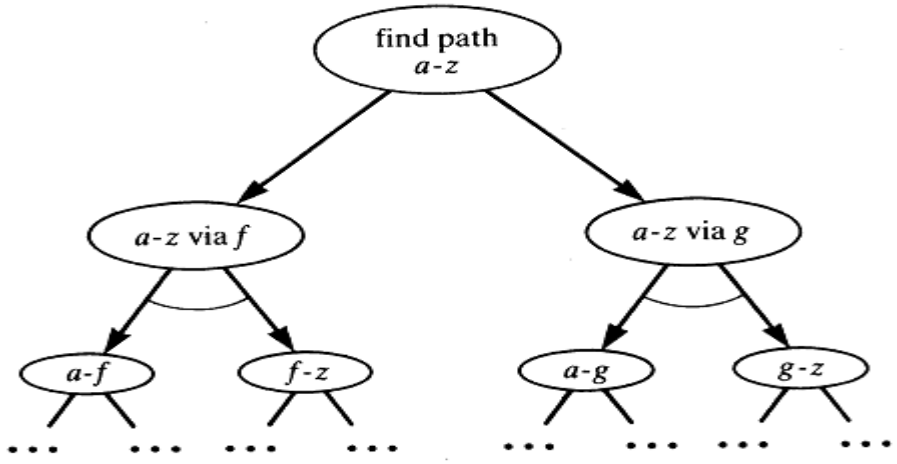
The problem could, of course, be formulated as path finding in a state space. The corresponding state space would look just like the map: the nodes in the state space correspond to cities, the arcs correspond to direct connections between cities, arc costs correspond to distances between cities. However, let us construct another representation of this problem, based on a natural decomposition of the problem.

In the map of Figure above, there is also a river. Let us assume that there are only two bridges at which the river can be crossed, one bridge at city ***f*** and the other at city ***g***. Obviously, our route will have to include one of the bridges; so it will have to go through ***f*** or through ***g***. We have, then, two major alternatives:

To find a path between ***a*** and ***z***, find either

(1) A path from ***a*** to ***z*** via ***f***, or

(2) A path from ***a*** to ***z*** via ***g***.
Each of these two alternative problems can now be decomposed as follows:
(1) To find a path from ***a*** to ***z*** via ***f***:
    1.1 Find a path from ***a*** to ***f***, and
    1.2 Find a path from ***f*** to ***z.***
(2) To find a path from ***a*** to ***z*** via ***g*:**
    2.1 Find a path from ***a*** to ***g***, and
    2.2 Find a path from ***g*** to ***z***.
The decomposition can be pictured as an AND/OR graph as in the figure (2) below:

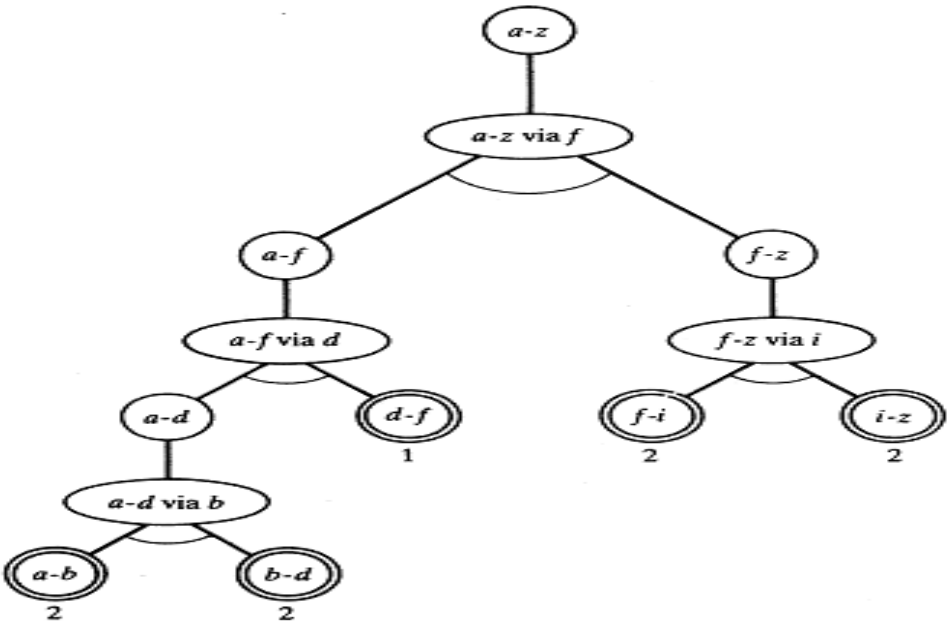


Notice the curved arcs which indicate the AND relationship between subproblems. Of course, the graph in Figure above is only the top part of a coresponding AND/OR tree. Further decomposition of subproblems could be based on the introduction of additional intermediate cities.
In principle, a node can issue both AND-related arcs and OR-related arcs. We will, however, assume that each node has either only AND successors or only OR successors. Each AND/OR graph can be transformed into this form by introducing auxiliary OR nodes if necessary. Then, a node that only issues AND arcs is called an AND node; a node that only issues OR arcs is called an OR node. In the state-space representation, a solution to the problem was a path in the state space while in the AND/OR representation, the solution has to include all the subproblems of an AND node.
For the shortest route problem of Figure (1), an AND/OR graph including a cost function can be defined as follows:

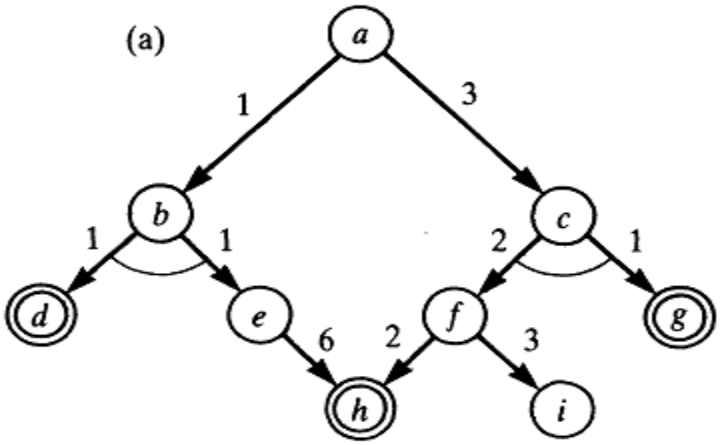
1- OR nodes are of the form **X-Z**, meaning: find a shortest path from **X** to **Z**.
2- AND nodes are of the form **X-Z** via **Y** meaning: find a shortest path from **X** to **Z** under the constraint that the path goes through **Y**.
3- A node **X-Z** is a goal node (primitive problem) if **X** and **Z** are directly connected in the map.
4- The cost of each goal node **X-Z** is the given road distance between **X** and **Z**.
5- The costs of all other (non-terminal) nodes are 0.

The cost of a solution graph is the sum of the costs of all the nodes in the solution graph (in our case, this is just the sum over the terminal nodes). Figure (3) below shows the best solution tree with a cost equal to 9. This tree corresponds to the path ***[a,b,d,f,i,z]***. This path can be reconstructed from the solution tree by visiting all the leaves in this tree in the left-to-right order.

The second route form ***a-z*** via ***g*** will cost 12.



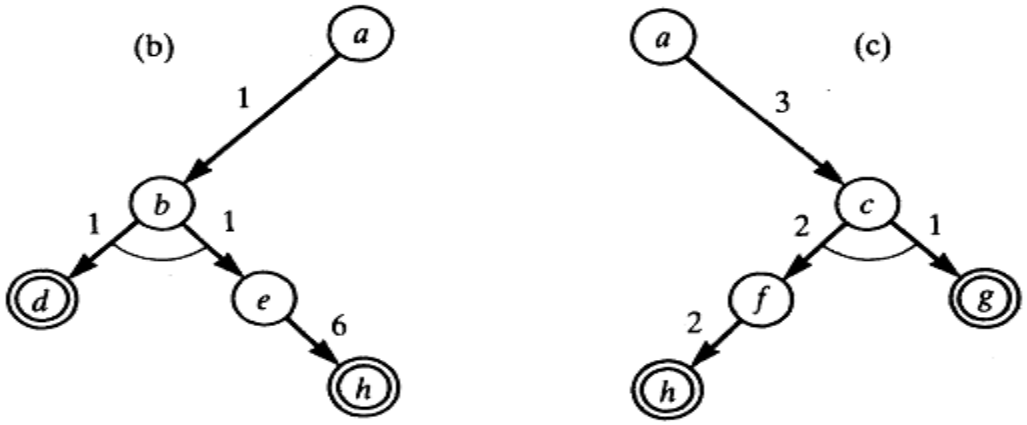
Example: In the figure (a) below, there are costs attached to arcs. The cost of a solution graph as the sum of all the arc costs in the graph.



Find all the solution graphs and then determine the solution graph with the minimum cost.

Solution:

As we are normally interested in the minimum cost, the solution graph in Figure (c) below will be preferred because it has summation cost equal to 8 and figure (b) is ignored because it has summation cost equal to 9.



AND/OR Graphs with Heuristic Estimates

The basic search procedures of the previous section search AND/OR graphs systematically and exhaustively, without any heuristic guidance. For complex problems such procedures are too inefficient due to the combinatorial complexity of the search space. The algorithm that using of heuristic guidance in AND/OR graph called AO* algorithm. For a node ***N*** in the search tree, ***H(N)*** will denote its estimated difficulty. For a tip node ***N*** of the current search tree, ***H(N)*** is simply ***h(N)*** . On the other hand, for an interior node of the search tree we do not have to use function ***h*** directly because we already have some additional information about such a node; that is, we already know its successors. For an interior OR node ***N*** we approximate its difficulty as:

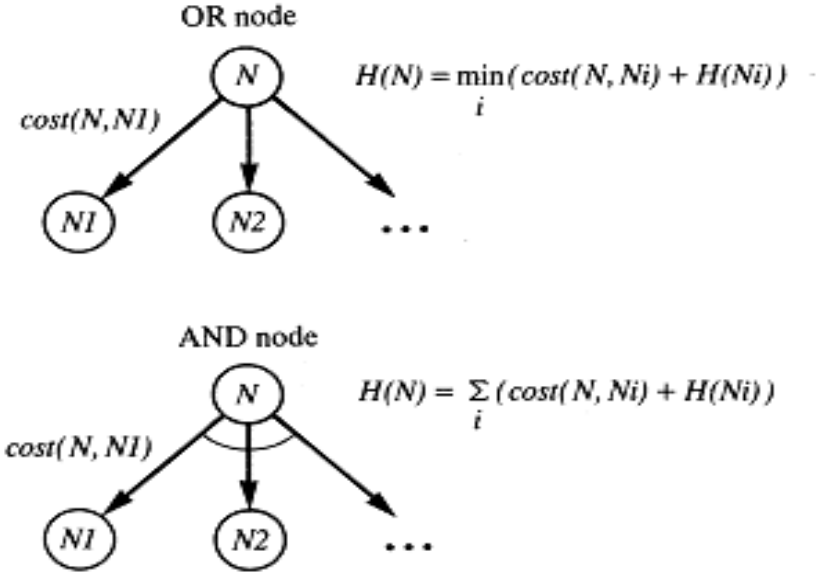
$$H(N) = \min_i (cost(N,N_i) + H(N_i))$$

where ***cost(N,Ni)*** is the cost of the arc from ***N*** to ***Ni***,. The minimization rule in this formula is justified by the fact that, to solve ***N***, we just have to solve one of its successors.

The difficulty of an AND node ***N*** is approximated by:

$$H(N) = \sum_i (cost(N,N_i) + H(N_i))$$

The two above functions are explained explicitly in the Figure (4) below:



In practice, another measure, **F**, defined in terms of **H**, instead of the H-values. Let a node **M** be the predecessor of **N** in the search tree, and the cost of the arc from **M** to **N** be **cost(M,N)**, then we define:

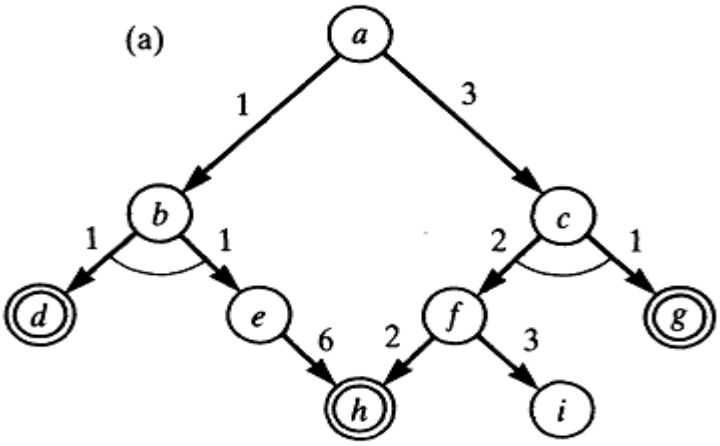
$$F(N) = cost(M,N) + H(N)$$

Accordingly, if M is the parent node of N, and $N_1, N_2, \ldots$ are N's children, then:

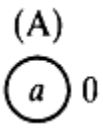
$$F(N) = cost(M,N) + \min_i F(N_i), \qquad \text{if } N \text{ is an OR node}$$

$$F(N) = cost(M,N) + \sum_i F(N_i), \qquad \text{if } N \text{ is an AND node}$$

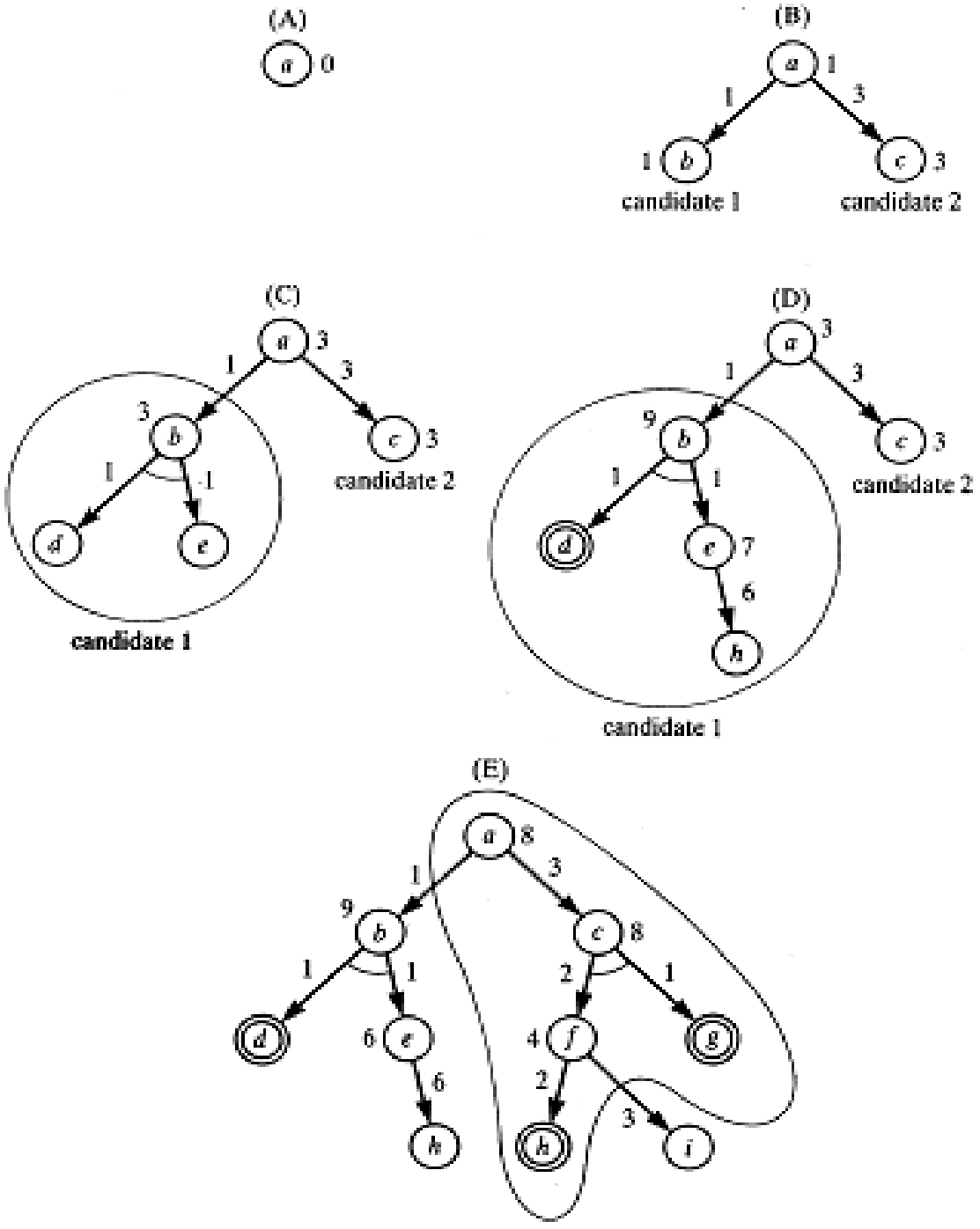
Example: Given the previous graph in Figure (a), that is:



Expand the initial search tree (**A**) using AO* algorithm.

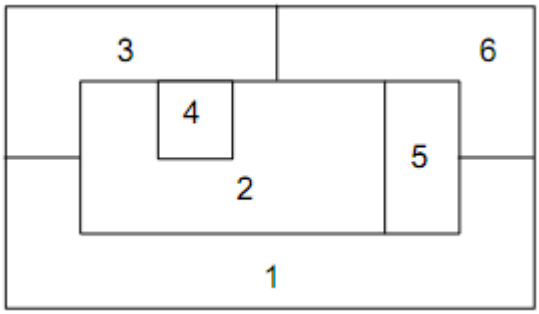


Solution: The initial search tree (**A**) will be expanded to produce four trees **B**, **C**, **D**, and **E** respectively as follows:



Constraint Satisfaction Problems

Constraint satisfaction problem (or **CSP**) is defined by a set of variables, **X1,X2, …, Xn**, and a set of constraints, **C1,C2, …, Cm**. Each variable Xi has a nonempty domain **Di** of possible values. Each constraint **Ci** involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, **{Xi = vi , Xj = vj ,…}**. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a **CSP** is a complete assignment that satisfies all the constraints. Some **CSPs** also require a solution that maximizes an objective function.
Example: Consider the following map. The task is to color the map using the four colors **Red**, **Blue**, **Yellow**, and **Green**, such that no two adjacent regions take the same color.



1. Formulate this problem as a **CSP**. Clearly state the variables, domains, and constraints.
2. Describe the topology of the constraint graph.
3. Color the map and Show the steps.

Solution:

1. The formulation of the problem as a **CSP** will be as follows:

a. **Variables={X1,X2,X3,X4,X5,X6}**

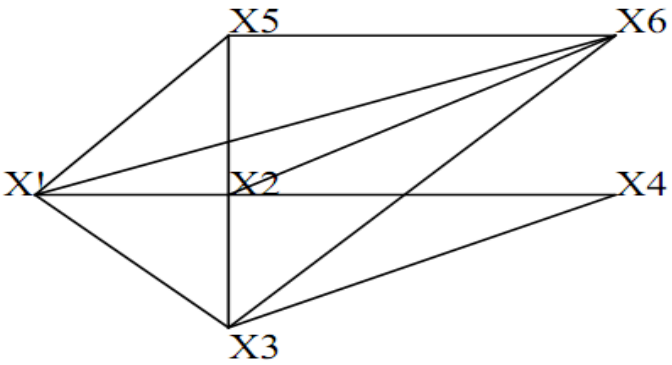
b. **Domain={Red,Blue,Yellow,Green}**

c. **Constraints=adjacent regions must have different colors.**

e.g.X1≠X2,X1≠X5,X1≠X6,X1≠X3, X2≠X4…

d. **Solutions are assignments satisfying all constraints**.

2. Constraint graph:



3. The steps of the algorithm is shown as follows:

Step1:

X1	X2	X3	X4	X5	X6
YRGB	YRGB	YRGB	YRGB	YRGB	YRGB

Step 2:

X1	X2	X3	X4	X5	X6
Y	RGB	RGB	YRGB	RGB	RGB

Step 3:

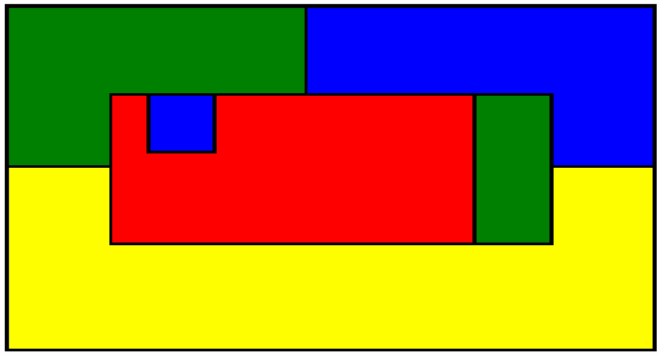
X1	X2	X3	X4	X5	X6
Y	R	GB	YGB	GB	GB

Step 4:

X1	X2	X3	X4	X5	X6
Y	R	G	YB	G B	B

Step 5:

X1	X2	X3	X4	X5	X6
Y	R	G	Y or B	G	B



Prolog Program for the Map Coloring Problem

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

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

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

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

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

Explanation:

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

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

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

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

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

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

Quick review for prolog programs (database and compound objects)

Database:

Example1:

1- Assert predicate:

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

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

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

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

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

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

2- **Retract predicate:**

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

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

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

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

```
domains
s=string.
ls=s*.
database
person(s)
predicates
list_preson(ls)
```

```
clauses
list_preson(L):-
        assert(person ("Ali")),
        assert(person ("Zaki")),
        assert(person ("Suha")),
        retractall(person (_)),% retractall(_),
        findall(X,person(X),L).

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

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

Solution:

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

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

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

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

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

Solution:

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

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

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

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

```
String=  bought    length=  6
String=  oranges    length=  7
String=  their    length=  5
String=  total    length=  5
String=  weight    length=  6
String=  kg    length=  2
yes
 */
```

Compound objects

Example1:

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

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

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

Example2:

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

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

Example3:

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

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

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

Example4:

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

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

goal
  member(s(a,5),[s(c,5),s(a,3),s(d,6)],X).%X=s("a",3)
  %member(s(a,3),[s(c,5),s(a,5),s(d,6)],X).%X=s("a",5)
  %member(s(d,1),[s(c,5),s(a,5),s(d,6)],X).%X=s("d",6)
  %member(s(c,1),[s(c,5),s(a,5),s(d,6)],X).%X=s("c",5)
```

Example5:

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

Example7:

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

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

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

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

References:

1. "Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig (2020, 4th Edition)
2. "Artificial Intelligence: Structures and Strategies for Complex Problem Solving" by George F. Luger (2021, 7th Edition)