Chapter 13
Problem Decomposition

In the previous two chapters, we discussed solving problems based on their state space representation. More specifically, once so represented, a solution for the problem is reduced to searching for a path from the start node to a solution in such a space.

In this chapter, we will talk about another way to represent problems, namely, And/Or graph, which suits certain class of problems particularly well.
An example

Consider the problem of finding a route in the following road map.

It is obvious that the above problem can be represented in the state space model, but since, the existence of the river makes the things a bit more restrictive. Thus, any path has to cross either the bridge in $f$, or $g$. 
A decomposition

Thus, to find a path from $a$ to $z$, we have the following two alternatives:

1. either finding a path via $f$; or
2. finding a path via $g$.

If we dig a bit deeper, we have the following decomposition of the problem:

1. *Either* 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. *or* find a path from $a$ to $z$ via $g$ :
   1.1. find a path from $a$ to $f$, and
   1.2. find a path from $g$ to $z$. 
Now, the graph

The above decomposition can be more vividly represented via the following *And/Or graph*:

In the above directed graph, each node stands for a problem, and arcs for relationships, particularly, the curved arc indicates the *And* relationship. Although in general, a node can have both And and Or arcs, we assume a node can have only one of them. Thus, the names of *And node*, and *Or node*. 
Solving the graph

A solution for a problem, $P$, represented in an And/Or graph is no longer a path, but rather a tree, which is a subgraph of the original And/Or graph. This tree is defined as the following $T$:

1. $P$ is the root of $T$.

2. If $P$ is an Or node, then exactly one of its successors together with its own solution tree, is included in $T$.

3. If $P$ is an And node, then all of its successors, together with their solution trees, are included in $T$. 
An example

Below shows an And/Or graph, in which $a$ is the problem, $d, h$ and $g$ are goal states, and the numbers attached to the arcs are costs. As we are usually interested in minimum-cost solutions, the one on the right is the preferred one.
Route finding

We can represent the shortest route problem as follows: We use two kinds of nodes: an Or node is of the form $X-Z$, meaning: find a shortest path from $X$ to $Z$; an And node is of the form $X-Z$ via $Y$: namely, find such a path via $Y$.

A node $X-Z$ is a goal node if $X$ is adjacent to $Z$, with the cost being the distance in between.

For all other non-terminal nodes, their costs are 0.
An example

We have that the total cost of a path from a to z is 9.
Hanoi Tower

This famous game can be described as follows: There are three pegs, 1, 2, and 3, and three disks $a, b,$ and $c$, in this order. Initially, all three disks are stacked in peg 1.

The problem is to transfer them all on to peg 3. Only one disk can be moved at a time, and no disk can even be placed on top of a smaller disk.

Below shows the initial, and the final, configuration of this game:
Solving the problem...

We really have to achieve a set of three goals: Disk \( a \) on peg 3; Disk \( b \) on peg 3; and Disk \( c \) on peg 3. But, they can not be achieved independently. For example, the first goal can be immediately done, but then will make the other two impossible.

Thus, what we have to do is to achieve the third goal first, i.e., somehow put \( c \) on peg 3, then try to satisfy the remaining goals. However, the third goal can not be achieved at once, we really have to do some “pre-processing.” Thus, a solution should look like the following:

1. Enable disk \( c \) to be moved from peg 1 to 3.
2. Move \( c \) from 1 to 3.
3. Achieve the other goals: both \( a \) and \( b \) on peg 3.
in terms of And/Or graph

Since $c$ can be moved from 1 to 3 only if both $a$ and $b$ are on peg 2. The above strategy becomes: to move $a,b$ and $c$ from peg 1 to 3:

1. Move $a$ and $b$ to peg 2, and
2. move $c$ from 1 to 3, and
3. move both $a$ and $b$ from peg 2 to 3.

Subgoal 2 is trivial, while the other two can be further decomposed. For example, the above 1 can be further decomposed as follows:

1.1. Move $a$ from 1 to 3, and
1.2. Move $b$ from 1 to 2, and
1.3. Move $a$ from 3 to 2.
Game playing in general

Games like chess and checkers can naturally be viewed as problems in terms of And/Or graphs. Such games are called *two-person, perfect information* games.

We further assume that there are only two outcomes, *Win*, or *Loss*, and we are the one to start. When starting, we have certain choices, and will win if *either* of these choices leads to a *Win*. For each of these choices, the opponent also has a few choices. If we win, we have to win for *all* of these choices. This naturally leads to the following And/Or graph.
Basic search strategies

The very search engine of Prolog is based on this And/Or mechanism. For example, given the following facts for an And/or graph:

\[
a: \neg b.
\]
\[
a: \neg c.
\]
\[
b: \neg d, e.
\]
\[
e: \neg h.
\]
\[
c: \neg f, g.
\]
\[
f: \neg h, i.
\]
\[
d. g. h.
\]

To see if the problem \( a \) can be solved, we simply say \(?-a.\)

It is very simple, but we only get an answer of “yes”, or “no”. It is also hard to extend it to handle cost information. Furthermore, it might enter an infinite loop when dealing with a general graph.
A few changes

We will introduce a binary relation \( \rightarrow \rightarrow \) so that the fact that there is an arc from \( a \), an or node, to \( b \) and \( c \) will be represented as \( a \rightarrow \rightarrow \text{or:} [b, c] \). Both \( \rightarrow \rightarrow \) and \( : \) can be defined as

\[
\text{:-op}(600, \text{xfx,} \rightarrow \rightarrow ).
\]
\[
\text{:-op}(500, \text{xfx,} :).
\]

Thus, the above facts can be restated as

\[
\begin{align*}
a & \rightarrow \rightarrow \text{or:} [b, c]. \\
b & \rightarrow \rightarrow \text{and:} [d, e]. \\
c & \rightarrow \rightarrow \text{and:} [f, g]. \\
e & \rightarrow \rightarrow \text{or:} [h]. \\
f & \rightarrow \rightarrow \text{or:} [h, i]. \\
\text{goal(d). goal(g). goal(h).}
\end{align*}
\]

**Question:** What is the nature of and and or in the above program?

**Answer:** They are merely arguments for the operator \( : \).
Now what?

Given this representation, we use the following principle to construct solution trees for a problem $N$.

1. If $N$ is a goal node, then it is trivially solved.

2. If $N$ has OR successors, then solve one of them by solving all the alternatives in turn.

3. If $N$ has AND successors, then solve all of them.

4. If the above rules do not lead to a solution, $N$ is not solvable.
In terms of Prolog

The above process can be described as follows:

\[
\begin{align*}
\text{solve(Node)}&:=-\text{goal(Node)}.
\text{solve(Node)}&:=-\text{Node}\rightarrow\text{or: Nodes},
\quad \text{member(Node1, Nodes)}, \text{solve(Node1)}.
\text{solve(Node)}&:=-\text{Node}\rightarrow\text{and: Nodes}, \text{solveall(Nodes)}.
\text{solveall([])}.\n\text{solveall([Node|Nodes])}&:- \text{solve(Node)},
\quad \text{solveall(Nodes)}.
\end{align*}
\]

This problem does not produce a solution tree, and it easily falls into an infinite loop.
Further improvement

To produce a solution tree, we add another argument SolutionTree into the solve relation. A solution tree can be represented as follows:

1. If Node is a goal node, then its solution tree contains just Node.

2. If Node is an or node, then its tree has the form: Node --> Subtree, where Subtree is a solution tree for one of its successors.

3. If it is an and node, then its solution tree has the form: Node --> and: Subtrees, where Subtrees is the list of the solution trees for all of its successors.

For example, one solution of a in the original tree can be this: a --> b --> and: [d, e --> h].
Obviously, the above three forms closely correspond to the three clauses in the program as given in the book, page 304, with the gist being the following:

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

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

solve( Node, Node ---> and:Trees) :-
    Node ---> and:Nodes,
    solveall( Nodes, Trees).

solveall( [], []).

solveall( [Node|Nodes], [Tree|Trees]) :-
    solve( Node, Tree),
    solveall( Nodes, Trees).
Further improvement

This revised program still easily gets into an infinite loop. One way to prevent this is to check the current path to see if its length is more than a preset limit. Thus, e.g.,

```prolog
solve(Node, Node--->Tree, MaxDepth):-
    MaxDepth>0, Node--->or:Nodes,
    member(Node1, Nodes),
    Depth1 is MaxDepth-1,
    solve(Node1, Tree, Depth1).
```

Homework: Complete Exercise 13.1, then apply it to various and/or graph, including the one in Figure 13.4, to look for solution trees.
Iterative deepening

The above depth challenged procedure can also be used in an iterative deepening way as follows:

iterative(Node, SolTree):-
    trydepth(Node, SolTree, 0).

trydepth(Node, SolTree, Depth):-
    solve(Node, SolTree, Depth)
;
    Depth1 is Depth+1,
    trydepth(Node, SolTree, Depth1).

Just as the same as its application in DFS, this iterative deepening approach always starts from the start when the limit is increased. On the other hand, it saves space when compared with the BFS strategy.

**Homework:** Think about 13.2.