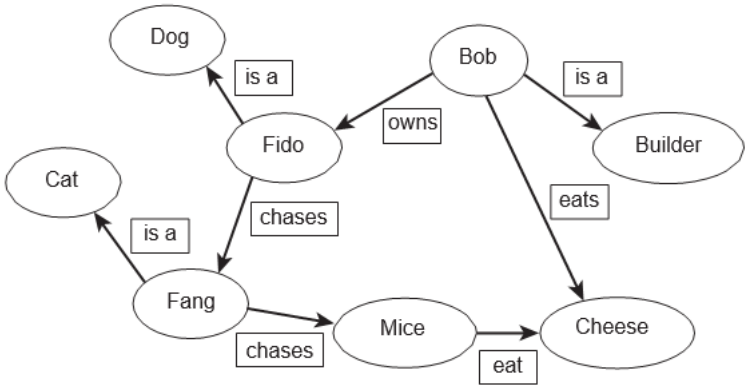
The Need for a Good Representation

When applying Artificial Intelligence to search problems, a useful, efficient, and meaningful representation is essential. In other words, the representation should be such that the computer does not waste too much time on pointless computations, it should be such that the representation really does relate to the problem that is being solved, and it should provide a means by which the computer can solve the problem.

Semantic Nets:

The semantic net is a commonly used representation in Artificial Intelligence. A semantic net is a **graph** consisting of **nodes** that are connected by **edges**. The nodes represent objects, and the links between nodes represent relationships between those objects. The links are usually labeled to indicate the nature of the relationship.

A simple example of a semantic net is shown in Figure 3.1.



Note that in this semantic net, the links are arrows, meaning that they have a direction. In this way, we can tell from the diagram that Fido chases Fang, not that Fang chases Fido. It may be that Fang does chase Fido as well, but this information is not presented in this diagram.

Semantic nets provide a very intuitive way to represent knowledge about objects and the relationships that exist between those objects. The data in semantic nets can be reasoned about to produce systems that have knowledge about a particular domain. Semantic nets do have limitations, such as the inability to represent negations: "Fido is not a cat."

Note that in our semantic net we have represented some specific individuals, such as Fang, Bob, and Fido, and have also represented some general classes of things, such as cats and dogs. The specific objects are generally referred to as instances of a particular class. Fido is an instance of the class dog. Bob is an instance of the class Builder.

It is a little unclear from the above Figure whether cheese is a class or an instance of a class. This information would need to be derived by the system that is manipulating the semantic net in some way. For example, the system might have a rule that says, "**any object that does not have an 'is-a' relationship to a class is considered to represent a class of objects**." Rules such as this must be applied with caution and must be remembered when building a semantic net.

An important feature of semantic nets is that they convey meaning. *That is to say, the relationship between nodes and edges in the net conveys information about some real-world situation.* **A good example of a semantic net is a family tree diagram. Usually, nodes in these diagrams represent people, and there are edges that represent parental relationships, as well as relation- ships by marriage.**

Each node in a semantic net has a label that identifies what the node represents. Edges are also labeled. Edges represent connections or relationships between nodes. In the case of searching a dictionary for a page that contains a particular word, each node might represent a single page, and each edge would represent a way of getting from one page to another.

The particular choice of semantic net representation for a problem will have great bearing on how the problem is solved. A simple representation for searching for a word in a dictionary would be to have the nodes arranged in a chain with one connection from the first node to the second, and then from the second to the third, and so on. Clearly, any method that attempts to search this graph will be fairly inefficient because it means visiting each node in turn until the desired node is found. This is equivalent to flicking through the pages of the dictionary in order until the desired page is found.

Inheritance:

Inheritance is a relationship that can be particularly useful in AI and in programming. The idea of inheritance is one that is easily understood intuitively. For example, if we say that all mammals give birth to live babies, and we also say that all dogs are mammals, and that Fido is a dog, then we can conclude that Fido gives birth to live mammals. Of course, this particular piece of reasoning does not take into account the fact that Fido might be male, or if Fido is female, might be too young or too old to give birth.

So, inheritance allows us to specify properties of a **superclass** and then to define a **subclass**, which inherits the properties of the superclass. In our example, mammals are the superclass of dogs and Fido. Dogs are the sub- class of mammals and the superclass of Fido.

As has been shown, although inheritance is a useful way to express generalities about a class of objects, in some cases we need to express exceptions to those generalities (such as, “Male animals do not give birth” or “Female dogs below the age of six months do not give birth”). In such cases, we say that the **default value** has been **overridden** in the subclass.

As we will see, it is usually useful to be able to express in our chosen representation which values can be overridden, and which cannot.

Frames:

Frame-based representation is a development of semantic nets and allows us to express the idea of inheritance.

As with semantic nets, a **frame system** consists of a set of frames (or nodes), which are connected together by relations. Each **frame** describes either an instance (an **instance frame**) or a class (a **class frame**).

Each frame has one or more **slots**, which are assigned **slot values**. This is the way in which the frame system network is built up. Rather than simply having links between frames, each relationship is expressed by a value being placed in a slot. For example, the semantic net in Figure 3.1 might be represented by the following frames:

Frame Name	Slot	Slot Value
Bob	is a	Builder
	owns	Fido
	eats	Cheese
Fido	is a	Dog
	chases	Fang
Fang	is a	Cat
	chases	Mice
Mice	eat	Cheese
Cheese		
Builder		
Dog		
Cat		

We can also represent this frame system in a diagrammatic form using representations such as those shown in Figure 3.2.

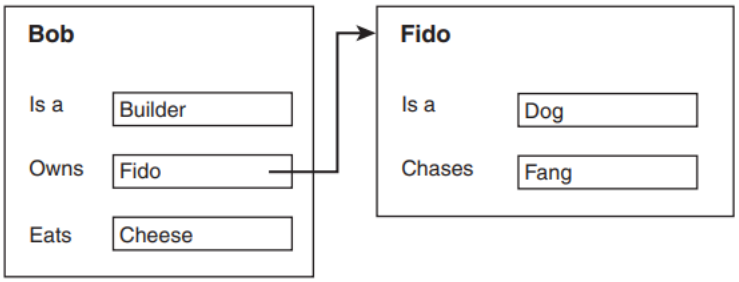


Figure 3.2
Partial representation for a frame system for the semantic net shown in Figure 3.1

When we say, "Fido is a dog," we really mean, "Fido is an instance of the class dog," or "Fido is a member of the class of dogs." Hence, the "is-a" relationship is very important in frame-based representations because it enables us to express membership of classes. This relationship is also known as **generalization** because referring to the class of mammals is more general than referring to the class of dogs and referring to the class of dogs is more general than referring to Fido.

It is also useful to be able to talk about one object being a part of another object. For example, Fido has a tail, and so the tail is part of Fido. This relationship is known as **aggregation** because Fido can be considered an aggregate of dog parts.

Other relationships are known as **association**. An example of such a relation- ship is the "chases" relationship. This explains how Fido and Fang are related or associated with each other. Note that association relationships have meaning in two directions. The fact that Fido chases Fang means that Fang is chased by Fido, so we are really expressing two relationships in one association.

Why Are Frames Useful?

The main advantage of using frame-based systems for expert systems over the rule-based approach is that all the information about a particular object is stored in one place. In a rule-based system, information about Fido might be stored in a number of otherwise unrelated rules, and so if Fido changes, or a deduction needs to be made about Fido, time may be wasted examining irrelevant rules and facts in the system, whereas with the frame system, the Fido frame could be quickly examined.

This difference becomes particularly clear when we consider frames that have a very large number of slots and where a large number of relationships exist between frames (i.e., a situation in which objects have a lot of properties, and a lot of objects are related to each other). Clearly, many real-world situations have these properties.

Inheritance

We might extend our frame system with the following additional information:

Dogs chase cats

Cats chase mice

In expressing these pieces of information, we now do not need to state explicitly that Fido chases Fang or that Fang chases mice. In this case, we can inherit this information because Fang is an instance of the class Cats, and Fido is an instance of the class Dogs.

We might also add the following additional information:

Mammals breathe

Dogs are mammals

Cats are mammals

Hence, we have now created a new superclass, mammals, of which dogs and cats are subclasses. In this way, we do not need to express explicitly that cats and dogs breathe because we can inherit this information. Similarly, we do not need to express explicitly that Fido and Fang breathe—they are instances of the classes Dogs and Cats, and therefore they inherit from those classes' superclasses.

Now let us add the following fact:

Mammals have four legs

Of course, this is not true, because humans do not have four legs, for example. In a frame-based system, we can express that this fact is the **default value** and that it may be overridden. Let us imagine that in fact Fido has had an unfortunate accident and now has only three legs. This information might be expressed as follows:

Frame Name	Slot	Slot Value
Mammal	*number of legs	four
Dog	subclass	Mammal
Cat	subclass	Mammal
Fido	is a	Dog
	number of legs	three
Fang	is a	Cat

Here we have used an asterisk (*) to indicate that the value for the "number of legs" slot for the Mammal class is a default value and can be overridden, as has been done for Fido.

Slots as Frames

It is also possible to express a range of values that a slot can take—for example, the number of legs slot might be allowed a number between 1 and 4 (although, for the insects' class, it might be allowed 6).

One way to express this kind of restriction is by allowing slots to be frames. In other words, the number of legs slot can be represented as a frame, which includes information about what range of values it can take:

Frame Name	Slot	Slot Value
Number of legs	minimum value	1
	maximum value	4

In this way, we can also express more complex ideas about slots, such as the **inverse** of a slot (e.g., the "chases" slot has an inverse, which is the "chased by" slot). We can also place further limitations on a slot, such as to specify whether it can take multiple values (e.g., the "number of legs" slot should probably only take one value, whereas the "eats" slot should be allowed to take many values).

Multiple Inheritance

It is possible for a frame to inherit properties from more than one other frame. In other words, a class can be a subclass of two superclasses, and an object can be an instance of more than one class. This is known as **multiple inheritance**.

For example, we might add the following frames to our system:

Frame Name	Slot	Slot Value
Human	Subclass	Mammal
	Number of legs	two
Builder	Builds	houses
Bob	is a	Human

From this, we can see that Bob is a human, as well as being a builder. Hence, we can inherit the following information about Bob:

He has two legs

He builds houses

In some cases, we will encounter conflicts, where multiple inheritance leads us to conclude contradictory information about a frame. For example, let us consider the following simple frame system:

Frame Name	Slot	Slot Value
Cheese	is	smelly
Thing wrapped in foil	is	not smelly
Cheddar	is a	Cheese
	is a	Thing wrapped in foil

(Note: the slot "is" might be more accurately named "has property." We have named it "is" to make the example clearer.)

Here we can see that cheddar is a type of cheese and that it comes wrapped in foil. Cheddar should inherit its smelliness from the Cheese class, but it also inherits non-smelliness from the Thing wrapped in foil class. In this case, we need a mechanism to decide which features to inherit from which superclasses. One simple method is to simply say that conflicts are resolved by the order in which they appear. So, if a fact is established by inheritance, and then that fact is contradicted by inheritance, the first fact is kept because it appeared first, and the contradiction is discarded.

This is clearly rather arbitrary, and it would almost certainly be better to build the frame system such that conflicts of this kind cannot occur.

Procedures

In object-oriented programming languages such as C++ or Java, classes (and hence objects) have **methods** associated with them. This is also true with frames. Frames have methods associated with them, which are called **procedures**. Procedures associated with frames are also called **procedural attachments**.

A procedure is a set of instructions associated with a frame that can be executed on request.

For example, a **slot reader** procedure might return the value of a particular slot within the frame. Another procedure might insert a value into a slot (a slot writer). Another important procedure is the instance constructor, which creates an instance of a class.

Such procedures are called when needed and so are called WHEN- NEEDED procedures. Other procedures can be set up that are called automatically when something changes.

Demons

A **demon** is a particular type of procedure that is run automatically when- ever a particular value changes or when a particular event occurs.

Some demons act when a particular value is read. In other words, they are called automatically when the user of the system, or the system itself, wants to know what value is placed in a particular slot. Such demons are called **WHEN-READ procedures**. In this way, complex calculations can be made that calculate a value to return to the user, rather than simply giving back static data that are contained within the slot. This could be useful, for example, in a large financial system with a large number of slots because it would mean that the system would not necessarily need to calculate every value for every slot. It would need to calculate some values only when they were requested.

WHEN-CHANGED procedures (also known as **WHEN-WRITTEN procedures**) are run automatically when the value of a slot is changed. This type of function can be particularly useful, for example, for ensuring that the values assigned to a slot fit within a set of constraints. For example, in our example above, a WHEN-WRITTEN procedure might run to ensure that the "number of legs" slot never has a value greater than 4 or less than 1. If a value of 7 is entered, a system message might be produced, telling the user that he or she has entered an incorrect value and that he or she should enter a different value.

Implementation

With the addition of procedures and demons, a frame system becomes a very powerful tool for reasoning about objects and relationships. The system has **procedural semantics** as opposed to **declarative semantics**, which means that the order in which things occur affects the results that the system produces. In some cases, this can cause problems and can make it harder to understand how the system will behave in a given situation.

This lack of clarity is usually compensated for by the level of flexibility allowed by demons and the other features that frame systems possess.

Frame systems can be implemented by a very simple algorithm if we do not allow multiple inheritance. The following algorithm allows us to find the value of a slot S, for a frame F. In this algorithm definition, we will use the notation F[S] to indicate the value of slot S in frame F. We also use the notation **instance (F1, F2)** to indicate that frame F1 is an instance of frame F2 and **subclass (F1, F2)** to indicate that frame F1 is a subclass of frame F2.

```
Function find_slot_value (S, F)
{
    if F[S] == V            // if the slot containsthen return
        V                   // a value, return it.
    else if instance (F, F')
        then return find_slot_value (S, F')else if
    subclass (F, F_S)
        then return find_slot_value (S, F_S)
    else return FAILURE;
}
```

In other words, the slot value of a frame F will either be contained within that frame, or a superclass of F, or another frame of which F is an instance. If none of these provides a value, then the algorithm fails.

If the system needs additional information to proceed, it can ask the user questions in order to fill in additional information. In the same way as with rule-based systems, WHEN-CHANGED procedures can be set up that monitor the values of slots, and when a particular set of values is identified, this can be used by the system to derive a conclusion and thus recommend an action or deliver an explanation for something.

Combining Frames with Rules

It is possible to combine frames with rules, and, in fact, many frame-based expert systems use rules in much the same way that rule-based systems do, with the addition of **pattern matching** clauses, which are used to identify values that match a set of conditions from all the frames in the system.

Typically, a frame-based system with rules will use rules to try to derive conclusions, and in some cases where it cannot find a value for a particular slot, a WHEN-NEEDED procedure will run to determine the value for that slot. If no value is found from that procedure, then the user will be asked to supply a value.

Representational Adequacy

We can represent the kinds of relationships that we can describe with frames in first-order predicate logic. For example:

$\forall x \text{ Dog } (x) \rightarrow \text{ Mammal } (x)$

First-order predicate logic is discussed in detail in Chapter 7. For now, you simply need to know how to read that expression. It is read as follows:

> "For all x's, if x is a dog, then x is a mammal."

This can be rendered in more natural English as:

> "All dogs are mammals."

In fact, we can also express this relationship by the introduction of a new symbol, which more closely mirrors the meaning encompassed by the idea of inheritance:

$$\text{Dog} \xrightarrow{\text{subset}} \text{Mammal}$$

Almost anything that can be expressed using frames can be expressed using first-order predicate logic (FPOL). The same is not true in reverse. For example, it is not easy to represent negativity ("Fido is not a cat") or quantification ("there is a cat that has only one leg"). We say that FOPL has greater **representational adequacy** than frame-based representations.

In fact, frame-based representations do have some aspects that cannot be easily represented in FOPL. The most significant of these is the idea of exceptions or overriding default values.

In this section, we have discussed three main representational methods: logic, rules, and frames (or semantic nets). Each of these has advantages and disadvantages, and each is preferable over the others in different situations. The important thing is that in solving a particular problem, the correct representation must be chosen.

Object-Oriented Programming

We now briefly explore some of the ideas used in object-oriented programming, and, in particular, we see how they relate to some of the ideas we have seen in Sections 3.4 and 3.5 on inheritance and frames.

Two of the best-known object-oriented programming languages are Java and C++. These two languages use a similar syntax to define classes and objects that are instantiations of those classes.

A typical class in these languages might be defined as:

```
class animal
{
        animal    ();   Eye
        *eyes; Leg  *legs;
        Head   head;  Tail
        tail;
}
```

This defines a class called animal that has a number of fields, which are the various body parts. It also has a **constructor**, which is a function that is called when an instantiation of the class is called. Classes can have other functions too, and these functions are equivalent to the procedures we saw in Section 3.5.5.

We can create an instance of the class animal as follows:

```
animal an_animal = new animal ();
```

This creates an instance of the class animal. The instance, which is an object, is called "an_animal". In creating it, the constructor animal () is called.

We can also create a subclass of animal:

```
Class dog : animal
{
          bark ();
}
```

Here we have created a subclass of animal called dog. Dog has inherited all of the properties of animal and also has a new function of its own called bark ().

In some object-oriented programming languages, it is possible to use multiple inheritance. This means that one class inherits properties from more than one parent class. While C++ does allow multiple inheritance, Java, which itself inherited many features from C++, does not allow multiple inheritance. This is because multiple inheritance was seen by the developers of Java as an "unclean" idea—one that creates unnecessarily complicated object-oriented structures. Additionally, it is always possible to achieve the same results using single inheritance as it is with multiple inheritance.

Object-oriented programming languages such as Java and C++ use the principles that were invented for the frames structure. There are also object-oriented programming languages such as IBM's APL2 that use a frame-based structure.

Search Spaces:

Many problems in Artificial Intelligence can be represented as search spaces. In simple terms, a search space is a representation of the set of possible choices in a given problem, one or more of which are the solution to the problem.

For example, attempting to find a particular word in a dictionary with 100 pages, a search space will consist of each of the 100 pages. The page that is being searched for is called a goal, and it can be identified by seeing

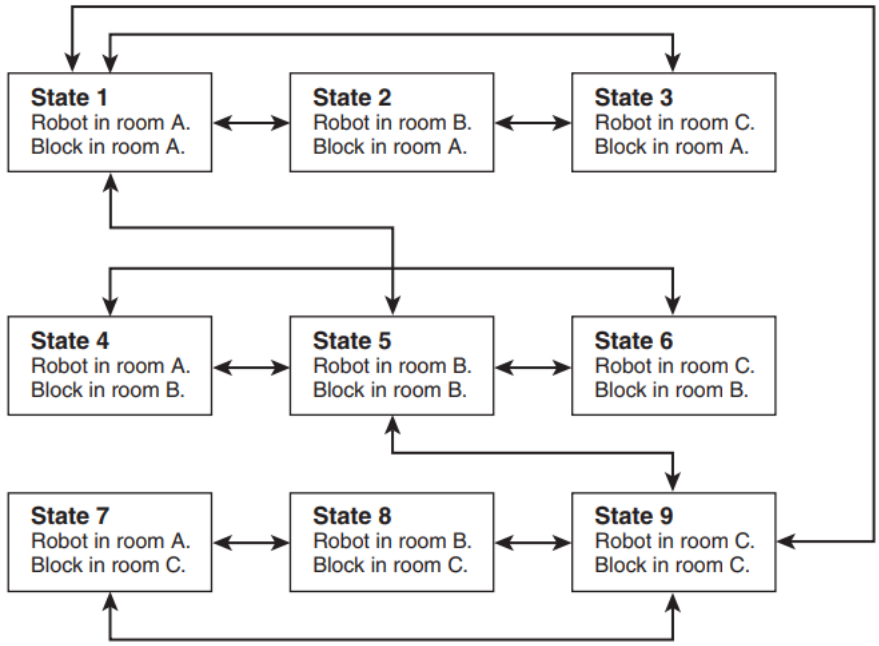


Figure 3.3
A simple state-space diagram

whether the word we are looking for is on the page or not. (In fact, this identification might be a search problem, but for this example we will assume that this is a simple, atomic action.)

The aim of most search procedures is to identify one or more goals and, usually, to identify one or more paths to those goals (often the shortest path, or path with least cost).

Because a search space consists of a set of states, connected by paths that represent actions, they are also known as **state spaces**. Many search problems can be represented by a state space, where the aim is to start with the world in one state and to end with the world in another, more desirable state. In the missionaries and cannibals problem that is discussed later in this chapter, the start state has all missionaries and cannibals on one side of the river, and the goal state has them on the other side. The state space for the problem consists of all possible states in between.

Figure above shows a very simple state-space diagram for a robot that lives in an environment with three rooms (room A, room B, and room C) and with a block that he can move from room to room. Each state consists of a possible arrangement of the robot and the block. Hence, for example, in state 1, both the robot and the block are in room A. Note that this diagram does not explain how the robot gets from one room to another or how the block is moved. This kind of representation assumes that the robot has a representation of a number of **actions** that it can take. To determine how to get from one state to another state, the robot needs to use a process called **planning**, which is covered in detail in Part 5 of this book.

In the above Figure, the arrows between states represent **state transitions**. Note that there are not transitions between every pair of states. For example, it is not possible to go from state 1 to state 4 without going through state 5. This is because the block cannot move on its own and can only be moved to a room if the robot moves there. Hence, a state-space diagram is a valuable way to represent the possible actions that can be taken in a given state and thus to represent the possible solutions to a problem.

Semantic Trees:

A semantic tree is a kind of semantic net that has the following properties:

- Each node (except for the root node, described below) has exactly one **predecessor** (parent) and one or more **successors** (children). In the semantic tree in Figure 3.4, node A is the predecessor of node B: node A connects by one edge to node B and comes before it in the tree. The successors of node B, nodes D and E, connect directly (by one edge each) to node B and come after it in the tree. We can write these relationships as: succ (B) = D and pred (B) = A.

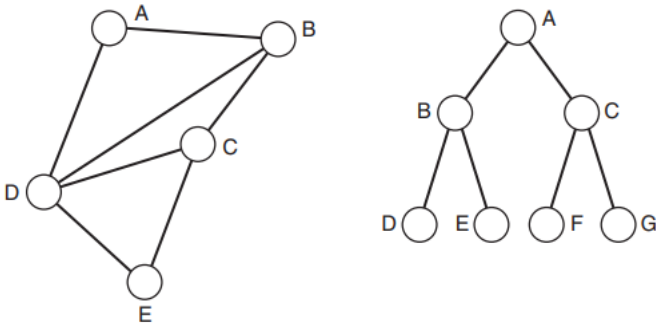


Figure 3.4
A semantic net and a semantic tree

The nonsymmetric nature of this relationship means that a semantic tree is a **directed** graph. By contrast, **nondirected graphs** are ones where there is no difference between an arc from A to B and an arc from B to A.

- One node has no predecessors. This node is called the root node. In general, when searching a semantic tree, we start at the root node. This is because the root node typically represents a starting point of the problem. For example, when we look at game trees in Chapter 6, we will see that the game tree for a game of chess represents all the possible moves of the game, starting from the initial position in which neither player has made a move. This initial position corresponds to the root node in the game tree.
- Some nodes have no successors. These nodes are called **leaf nodes**. One or more leaf nodes are called **goal nodes**. These are the nodes that represent a state where the search has succeeded.
- Apart from leaf nodes, all nodes have one or more successors. Apart from the root node, all nodes have exactly one predecessor.
- An **ancestor** of a node is a node further up the tree in some path. A **descendent** comes after a node in a path in the tree.

A **path** is a route through the semantic tree, which may consist of just one node (a path of length 0). A path of length 1 consists of a node, a branch that leads from that node, and the successor node to which that branch leads. A path that leads from the root node to a goal node is called a **complete path**. A path that leads from the root node to a leaf node that is not a goal node is called a **partial path**.

When comparing semantic nets and semantic trees visually, one of the most obvious differences is that semantic nets can contain cycles, but semantic trees cannot. A **cycle** is a path through the net that visits the same node more than once. Figure above shows a semantic net and a semantic tree. In the semantic net, the path A, B, C, D, A.. is a cycle.

In semantic trees, an edge that connects two nodes is called a **branch**. If a node has *n* successors, that node is said to have a **branching factor** of *n*. A tree is often said to have a branching factor of *n* if the average branching factor of all the nodes in the tree is *n*.

The root node of a tree is said to be at level 0, and the successors of the root node are at level 1. Successors of nodes at level n are at level $n + 1$.

Search Trees:

Searching a semantic net involves traversing the net systematically (or in some cases, not so systematically), examining nodes, looking for a goal node. Clearly following a cyclic path through the net is pointless because following A, B, C, D, A will not lead to any solution that could not be reached just by starting from A. We can represent the possible paths through a semantic net as a **search tree**, which is a type of semantic tree.

The search tree shown in Figure 3.5 represents the possible paths through the semantic net shown in Figure 3.4. Each node in the tree represents a path, with successive layers in the tree representing longer and longer paths. Note that we do not include cyclical paths, which means that some branches in the search tree end on leaf nodes that are not goal nodes. Also note that we label each node in the search tree with a single letter, which represents the path from the root node to that node in the semantic net in Figure 3.4.

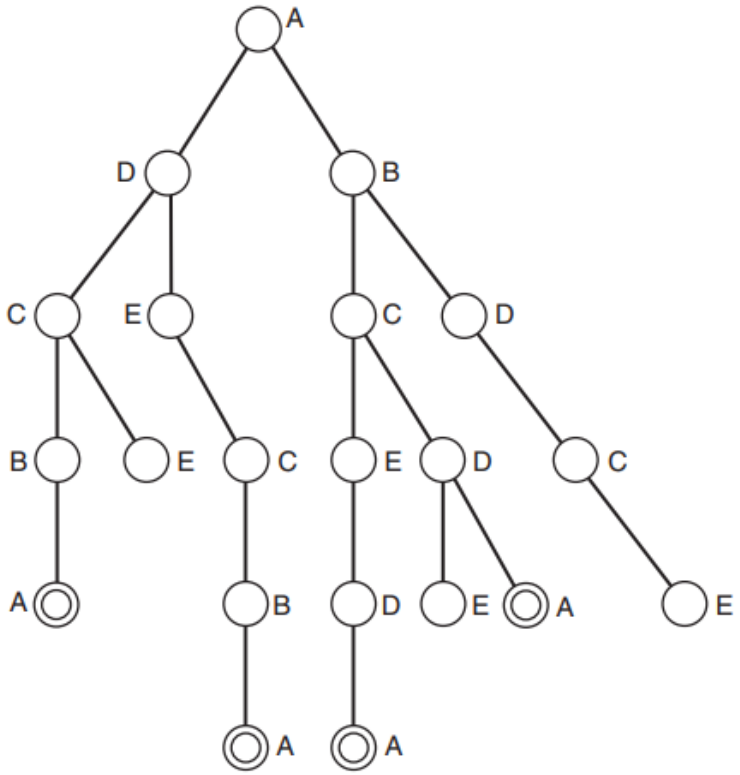


Figure 3.5
A search tree representation for the semantic net in Figure 3.4.

Hence, searching for a node in a search tree corresponds to searching for a complete path in a semantic net.

Example 1: Missionaries and Cannibals

The Missionaries and Cannibals problem is a well-known problem that is often used to illustrate AI techniques. The problem is as follows:

Three missionaries and three cannibals are on one side of a river, with a canoe. They all want to get to the other side of the river. The canoe can only hold one or two people at a time. At no time should there be more cannibals than missionaries on either side of the river, as this would probably result in the missionaries being eaten.

To solve this problem, we need to use a suitable representation.

First, we can consider a state in the solving of the problem to consist of a certain number of cannibals and a certain number of missionaries on each side of the river, with the boat on one side or the other. We could represent this, for example, as

3, 3, 1 0, 0, 0

The left-hand set of numbers represents the number of cannibals, missionaries, and canoes on one side of the river, and the right-hand side represents what is on the other side.

Because the number that is on one side is entirely dependent on the number that is on the other side, we can in fact just show how many of each are on the finishing side, meaning that the starting state is represented as

0, 0, 0

and the goal state is

3, 3, 1

An example of a state that must be avoided is

2, 1, 1

Here, there are two cannibals, one canoe, and just one missionary on the other side of the river. This missionary will probably not last very long.

To get from one state to another, we need to apply an operator. The operators that we have available are the following:

1. Move one cannibal to the other side
2. Move two cannibals to the other side
3. Move one missionary to the other side
4. Move two missionaries to the other side
5. Move one cannibal and one missionary to the other side

So, if we apply operator 5 to the state represented by 1, 1, 0, then we would result in state 2, 2, 1. One cannibal, one missionary, and the canoe have now moved over to the other side. Applying operator 3 to this state would lead to an illegal state: 2, 1, 0.

We consider rules such as this to be **constraints**, which limit the possible operators that can be applied in each state. If we design our representation correctly, the constraints are built in, meaning we do not ever need to examine illegal states.

We need to have a test that can identify if we have reached the goal state - 3, 3, 1.

We will consider the cost of the path that is chosen to be the number of steps that are taken, or the number of times an operator is applied. In some cases, as we will see later, it is desirable to find a solution that minimizes cost.

The first three levels of the search tree for the missionaries and cannibals problem is shown in Figure 3.6 (arcs are marked with which operator has been applied).

Now, by extending this tree to include all possible paths, and the states those paths lead to, a solution can be found. A solution to the problem would be represented as a path from the root node to a goal node.

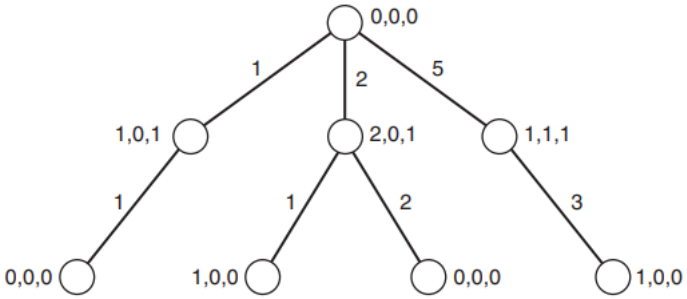


Figure 3.6
A partial search tree for the missionaries and cannibals problem

This tree represents the presence of a cycle in the search space. Note that the use of search trees to represent the search space means that our representation never contains any cycles, even when a cyclical path is being followed through the search space.

By applying operator 1 (moving one cannibal to the other side) as the first action, and then applying the same operator again, we return to the start state. This is a perfectly valid way to try to solve the problem, but not a very efficient one.

Improving the Representation

A more effective representation for the problem would be one that did not include any cycles. Figure 3.7 is an extended version of the search tree for the problem that omits cycles and includes goal nodes.

Note that in this tree, we have omitted most repeated states. For example, from the state 1,0,0, operator 2 is the only one shown. In fact, operators 1 and 3 can also be applied, leading to states 2,0,1 and 1,1,1 respectively. Neither of these transitions is shown because those states have already appeared in the tree.

As well as avoiding cycles, we have thus removed suboptimal paths from the tree. If a path of length 2 reaches a particular state, s, and another path of length 3 also reaches that state, it is not worth pursuing the longer path because it cannot possibly lead to a shorter path to the goal node than the first path.

Hence, the two paths that can be followed in the tree in Figure 3.7 to the goal node are the shortest routes (the paths with the least cost) to the goal, but they are by no means the only paths. Many longer paths also exist.

By choosing a suitable representation, we are thus able to improve the efficiency of our search method. Of course, in actual implementations, things may not be so simple. To produce the search tree without repeated states, a memory is required that can store states in order to avoid revisiting them. It is likely that for most problems this memory requirement is a worthwhile tradeoff for the saving in time, particularly if the search space being explored has many repeated states and cycles.

Solving the Missionaries and Cannibals problem involves **searching** the search tree. As we will see, search is an extremely useful method for solving problems and is widely used in Artificial Intelligence.

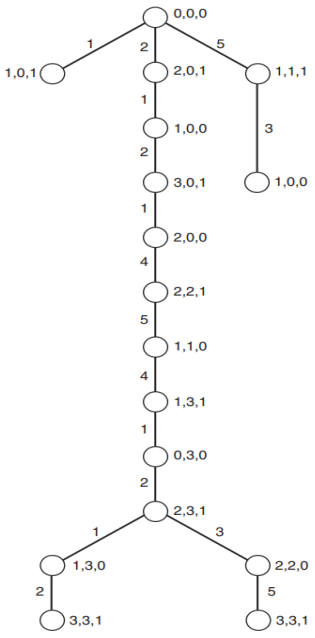


Figure 3.7
Search tree without cycles

Example 2: The Traveling Salesman

The Traveling Salesman problem is another classic problem in Artificial Intelligence and is **NP-Complete**, meaning that for large instances of the problem, it can be very difficult for a computer program to solve in a rea- sonable period of time. A problem is defined as being in the class P if it can be solved in polynomial time. This means that as the size of the problem increases, the time it will take a deterministic computer to solve the problem will increase by some polynomial function of the size. Problems that are NP can be solved non-deterministically in polynomial time. This means that if a possible solution to the problem is presented to the computer, it will be able to determine whether it is a solution or not in polynomial time. The hardest NP problems are termed NP-Complete. It was shown by Stephen Cook that a particular group of problems could be transformed into the satisfiability problem (see Chapter 16). These problems are defined as being NP-Complete. This means that if one can solve the satisfiability problem (for which solutions certainly do exist), then one can solve any NP-Complete problem. It also means that NP-Complete problems take a great deal of computation to solve.

The Traveling Salesman problem is defined as follows:

A salesman must visit each of a set of cities and then return home. The aim of the problem is to find the shortest path that lets the salesman visit each city.

Let us imagine that our salesman is touring the following American cities:

A Atlanta

B Boston

C Chicago

D Dallas

E El Paso

Our salesman lives in Atlanta and must visit all of the other four cities before returning home. Let us imagine that our salesman is traveling by plane and that the cost of each flight is directly proportional to distance being traveled and that direct flights are possible between any pair of cities.

Hence, the distances can be shown on a graph as in Figure 3.8.

(Note: The distances shown are not intended to accurately represent the true locations of these cities but have been approximated for the purposes of this illustration.)

The graph in Figure 3.8 shows the relationships between the cities. We could use this graph to attempt to solve the problem. Certainly, we can use it to find possible paths: One possible path is A, B, C, E, D, A, which has a length of 4500 miles.

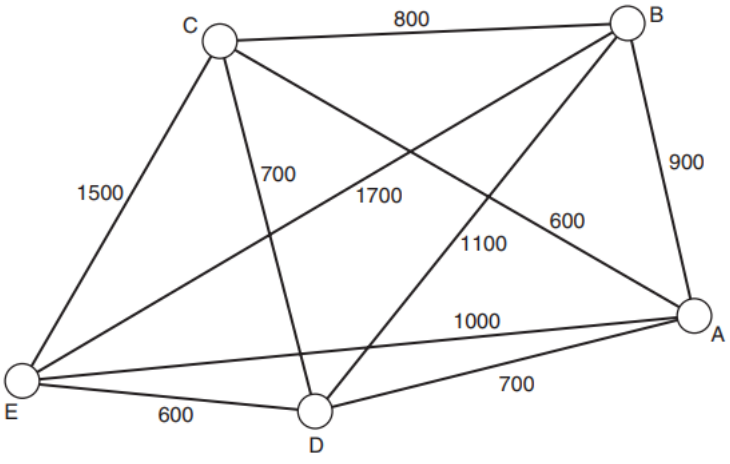


Figure 3.8
Simplified map showing Traveling Salesman problem with five cities

To solve the problem using search, a different representation would be needed, based on this graph. Figure 3.9 shows a part of the search tree that represents the possible paths through the search space in this problem. Each node is marked with a letter that represents the city that has been reached by the path up to that point. Hence, in fact, each node represents the path from city A to the city named at that node. The root node of the graph thus represents the path of length 0, which consists simply of the city A. As with the previous example, cyclical paths have been excluded from the tree, but unlike the tree for the missionaries and cannibals problem, the tree does allow repeated states. This is because in this problem each state must be visited once, and so a complete path must include all states. In the Missionaries and Cannibals problem, the aim was to reach a particular state by the shortest path that could be found. Hence, including a path such as A, B, C, D where a path A, D had already been found would be wasteful because it could not possibly lead to a shorter path than A, D. With the Traveling Salesman problem, this does not apply, and we need to examine every possible path that includes each node once, with the start node at the beginning and the end.

Figure 3.9 is only a part of the search tree, but it shows two complete paths: A, B, C, D, E, A and A, B, C, E, D, A. The total path costs of these two paths are 4000 miles and 4500 miles, respectively.

In total there will be (n — 1)! possible paths for a Traveling Salesman problem with n cities. This is because we are constrained in our starting city and, thereafter, have a choice of any combination of (n — 1) cities. In problems with small numbers of cities, such as 5 or even 10, this means that the complete search tree can be evaluated by a computer program without much difficulty; but if the problem consists of 40 cities, there would be 40! paths, which is roughly 1048, a ludicrously large number. As we see in the next chapter, methods that try to examine all of these paths are called **brute-force search** methods. To solve search problems with large trees, knowledge about the problem needs to be applied in the form of **heuristics**, which enable us to find more efficient ways to solve the problem. A heuristic is a rule or piece of information that is used to make search or another problem-solving method more effective or more efficient.

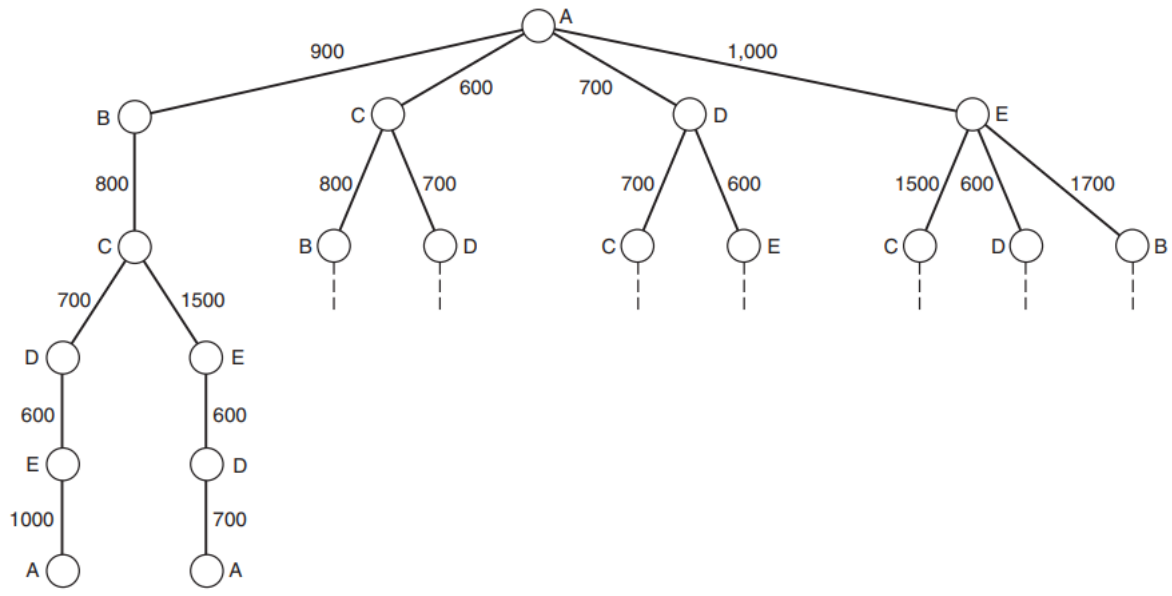


Figure 3.9
Partial search tree for Traveling Salesman problem with five cities

For example, a heuristic search approach to solving the Traveling Salesman problem might be: rather than examining every possible path, we simply extend the path by moving to the city closest to our current position that has not yet been examined. This is called the **nearest neighbor heuristic**. In our example above, this would lead to the path A,C,D,E,B,A, which has a total cost of 4500 miles. This is certainly not the best possible path, as we have already seen one path (A, B, C, D, E, A) that has a cost of 4000 miles. This illustrates the point that although heuristics may well make search more efficient, they will not necessarily give the best results. We will see methods in the next chapters that illustrate this and will also discuss ways of choosing heuristics that usually do give the best result.

Example 3: The Towers of Hanoi

The Towers of Hanoi problem is defined as follows:

We have three pegs and a number of disks of different sizes. The aim is to move from the starting state where all the disks are on the first peg, in size order (smallest at the top) to the goal state where all the pegs are on the third peg, also in size order. We are allowed to move one disk at a time, as long as there are no disks on top of it, and as long as we do not move it on top of a peg that is smaller than it.

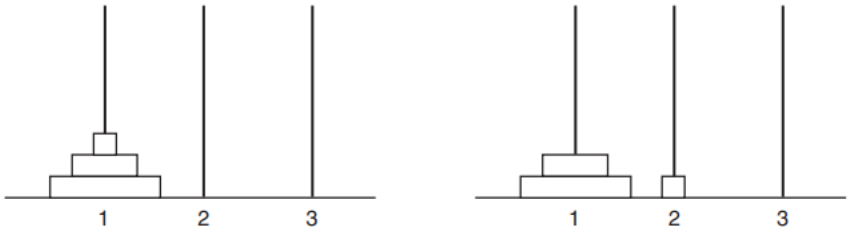


Figure 3.10
Two states in the Towers of Hanoi problem

Figure above shows the start state and a state after one disk has been moved from peg 1 to peg 2 for a Towers of Hanoi problem with three disks.

Now that we know what our start state and goal state look like, we need to come up with a set of operators:

Op1 Move disk from peg 1 to peg 2

Op2 Move disk from peg 1 to peg 3

Op3 Move disk from peg 2 to peg 1

Op4 Move disk from peg 2 to peg 3

Op5 Move disk from peg 3 to peg 1

Op6 Move disk from peg 3 to peg 2

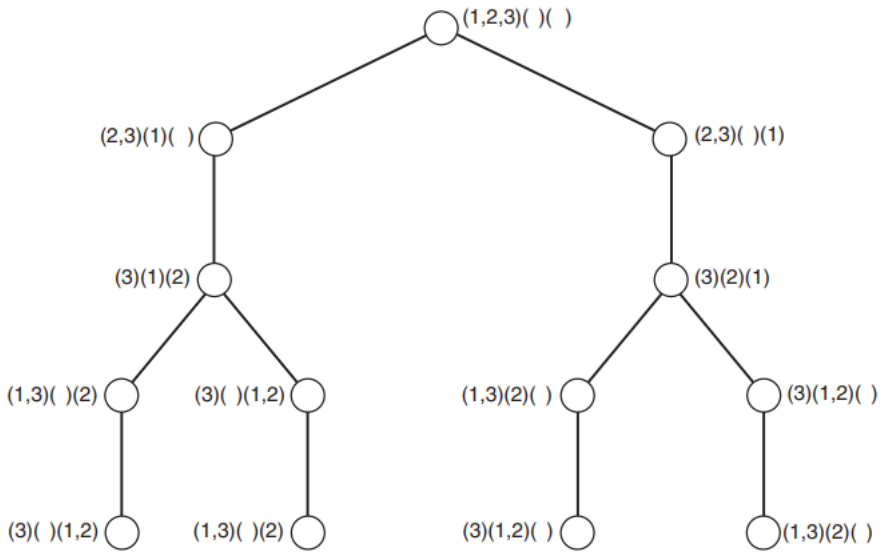


Figure 3.11
The first five levels of the search tree for the Towers of Hanoi problem with three disks

We also need a way to represent each state. For this example, we will use vectors of numbers where 1 represents the smallest peg and 3 the largest peg. The first vector represents the first peg, and so on. Hence, the starting state is represented as

(1,2,3) () ()

The second state shown in figure 3.10 is represented as

(2,3) (1) ()

and the goal state is () () (1,2,3)

The first few levels of the search tree for the Towers of Hanoi problem with three disks is shown in Figure 3.11. Again, we have ignored cyclical paths. In fact, with the Towers of Hanoi problem, at each step, we can always choose to reverse the previous action. For example, having applied opera- tor Op1 to get from the start state to (2,3) (1) (), we can now apply opera- tor Op3, which reverses this move and brings us back to the start state. Clearly, this behavior will always lead to a cycle, and so we ignore such choices in our representation.

As we see later in this book, search is not the only way to identify solutions to problems like the Towers of Hanoi. A search method would find a solution by examining every possible set of actions until a path was found that led from the start state to the goal state. A more intelligent system might be developed that understood more about the problem and, in fact, under- stood how to go about solving the problem without necessarily having to examine any alternative paths at all.

Describe and Match

A method used in Artificial Intelligence to identify objects is to describe it and then search for the same description in a database, which will identify the object.

An example of Describe and Match is as follows:

Alice is looking out of her window and can see a bird in the garden. She does not know much about birds but has a friend, Bob, who does. She calls Bob and describes the bird to him. From her description, he can tell her that the bird is a penguin.

We could represent Bob's knowledge of birds in a search tree, where each node represents a question, and an arc represents an answer to the question. A path through the tree describes various features of a bird, and a leaf node identifies the bird that is being described.

Hence, Describe and Match enables us to use search in combination with knowledge to answer questions about the world.

A portion of the search tree Bob used to identify the penguin outside Alice's window is shown in Figure 3.12.

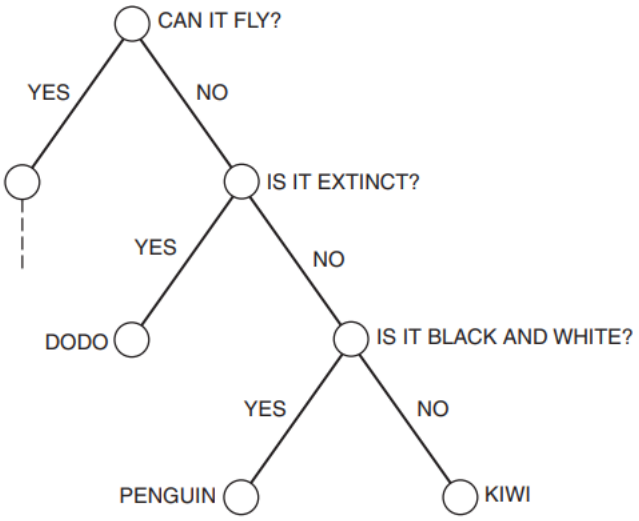


Figure 3.12
Search tree representation used with Describe and Match to identify a penguin

First, the question at the top of the tree, in the root node, is asked. The answer determines which branch to follow from the root node. In this case, if the answer is "yes," the left-hand branch is taken (this branch is not shown in the diagram). If the answer is "no," then the right-hand branch is taken, which leads to the next question—"Is it extinct?"

If the answer to this question is "yes," then a leaf node is reached, which gives us the answer: the bird is a dodo. If the answer is "no," then we move on to the next question. The process continues until the algorithm reaches a leaf node, which it must eventually do because each step moves one level down the tree, and the tree does not have an infinite number of levels.

This kind of tree is called a **decision tree**, and we learn more about them in Chapter 10, where we see how they are used in machine learning.

Combinatorial Explosion

The search tree for a Traveling Salesman problem becomes unmanageably large as the number of cities increases. Many problems have the property that as the number of individual items being considered increases, the number of possible paths in the search tree increases **exponentially**, meaning that as the problem gets larger, it becomes more and more unreason- able to expect a computer program to be able to solve it. This problem is known as **combinatorial explosion** because the amount of work that a program needs to do to solve the problem seems to grow at an explosive rate, due to the possible combinations it must consider.

Problem Reduction

In many cases we find that a complex problem can be most effectively solved by breaking it down into several smaller problems. If we solve all of those smaller **subproblems**, then we have solved the main problem. This approach to problem solving is often referred to as **goal reduction** because it involves considering the ultimate goal of solving the problem in a way that involves generating subgoals for that goal.

For example, to solve the Towers of Hanoi problem with n disks, it turns out that the first step is to solve the smaller problem with n — 1 disks.

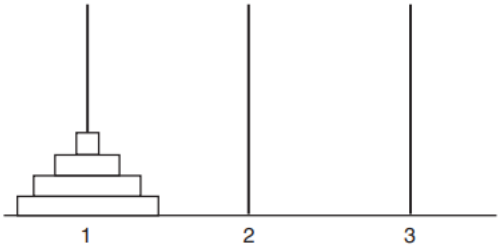


Figure 3.13
The starting state of the Towers of Hanoi problem with four disks

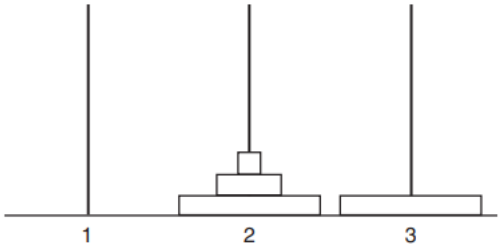


Figure 3.14
Towers of Hanoi problem of size 4 reduced to a problem of size 3 by first moving the largest disk from peg 1 to peg 3

For example, let us examine the Towers of Hanoi with four disks, whose starting state is shown in Figure 3.13.

To solve this problem, the first step is to move the largest block from peg 1 to peg 3. This will then leave a Towers of Hanoi problem of size 3, as shown in Figure 3.14, where the aim is to move the disks from peg 2 to peg 3. Because the disk that is on peg 3 is the largest disk, any other disk can be placed on top of it, and because it is in its final position, it can effectively be ignored.

In this way, a Towers of Hanoi problem of any size n can be solved by first moving the largest disk to peg 3, and then applying the Towers of Hanoi solution to the remaining disks but swapping peg 1 and peg 2.

The method for moving the largest disk is not difficult and is left as an exercise.

Goal Trees

A **goal tree** (also called an **and-or tree**) is a form of semantic tree used to represent problems that can be broken down in this way. We say that the solution to the problem is the **goal**, and each individual step along the way is a **subgoal**. In the case of the Towers of Hanoi, moving the largest disk to peg 3 is a subgoal.

Each node in a goal tree represents a subgoal, and that node's children are the subgoals of that goal. Some goals can be achieved only by solving all of its subgoals. Such nodes on the goal tree are **and-nodes**, which represent **and-goals**.

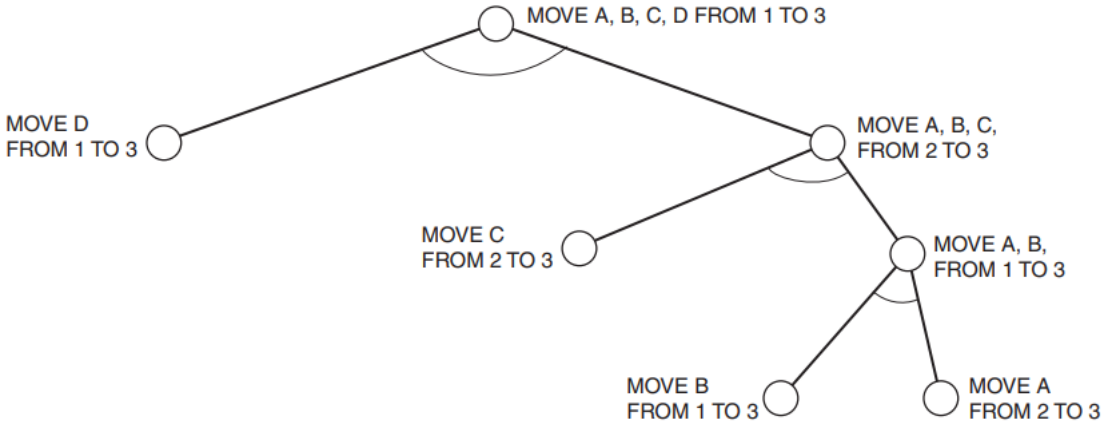


Figure 3.15
Goal tree for Towers of Hanoi problem with four disks

In other cases, a goal can be achieved by achieving any one of its subgoals. Such goals are **or-goals** and are represented on the goal tree by **or-nodes**.

Goal trees are drawn in the same way as search trees and other semantic trees. An and-node is shown by drawing an arc across the arcs that join it to its subgoals (children). Or-nodes are not marked in this way. The main difference between goal trees and normal search trees is that in order to solve a problem using a goal tree, a number of subproblems (in some cases, all subproblems) must be solved for the main problem to be solved. Hence, leaf nodes are called **success nodes** rather than goal nodes because each leaf node represents success at a small part of the problem.

Success nodes are always and-nodes. Leaf nodes that are or-nodes are impossible to solve and are called **failure nodes**.

A goal tree for the Towers of Hanoi problem with four disks is shown in Figure 3.15. The root node represents the main goal, or **root goal**, of the problem, which is to move all four disks from peg 1 to peg 3. In this tree, we have represented the four disks as A, B, C and D, where A is the smallest disk, and D is the largest. The pegs are numbered from 1 to 3. All of the nodes in this tree are and-nodes. This is true of most problems where there is only one reasonable solution.

Figure 3.15 is somewhat of an oversimplification because it does not explain how to solve each of the subgoals that is presented. To produce a system that could solve the problem, a larger goal tree that included additional subgoals would be needed. This is left as an exercise.

Breaking down the problem in this way is extremely advantageous because it can be easily extended to solving Towers of Hanoi problems of all sizes. Once we know how to solve the Towers of Hanoi with three disks, we then know how to solve it for four disks. Hence, we also know how to solve it for five disks, six disks, and so on. Computer programs can be developed easily that can solve the Towers of Hanoi problem with enormous numbers of disks.

Another reason that reducing problems to subgoals in this way is of such great interest in Artificial Intelligence research is that this is the way in which humans often go about solving problems. If you want to cook a fancy dinner for your friends, you probably have a number of subgoals to solve first:

- find a recipe
- go to the supermarket
- buy ingredients
- cook dinner
- set the table

And so on. Solving the problem in this way is very logical for humans because it treats a potentially complex problem as a set of smaller, simpler problems. Humans work very well in this way, and in many cases computers do too.

One area in which goal trees are often used is computer security. A **threat tree** represents the possible threats to a computer system, such as a computerized banking system. If the goal is "steal Edwin's money from the bank," you can (guess **or** convince me to divulge my PIN) **and** (steal **or** copy my card) and so on. The threat tree thus represents the possible paths an attacker of the system might take and enables security experts to deter- mine the weaknesses in the system.

Top Down or Bottom Up?

There are two main approaches to breaking down a problem into sub- goals—**top down** and **bottom up.**

A top-down approach involves first breaking down the main problem into smaller goals and then recursively breaking down those goals into smaller goals, and so on, until leaf nodes, or success nodes, are reached, which can be solved.

A bottom-up approach involves first determining all of the subgoals that are necessary to solve the entire problem, and then starting by solving the success nodes, and working up until the complete solution is found. As we see elsewhere in this book, both of these approaches are valid, and the correct approach should be taken for each problem.

Again, humans often think in these terms.

Businesses often look at solving problems either from the top down or from the bottom up. Solving a business problem from the top-down means looking at the global picture and working out what subgoals are needed to change that big picture in a satisfactory way. This often means passing those subgoals onto middle managers, who are given the task of solving them. Each middle manager will then break the problem down into smaller subproblems, each of which will be passed down the chain to subordinates. In this way, the overall problem is solved without the senior management ever needing to know how it was actually solved. Individual staff members solve their small problems without ever knowing how that impacts on the overall business.

A bottom-up approach to solving business problems would mean looking at individual problems within the organization and fixing those. Computer systems might need upgrading, and certain departments might need to work longer hours. The theory behind this approach is that if all the individual units within the business are functioning well, then the business as a whole will be functioning well.

Uses of Goal Trees

We can use goal-driven search to search through a goal tree. As we describe elsewhere in this book, this can be used to solve a number of problems in Artificial Intelligence.

Example 1: Map Coloring

Map-coloring problems can be represented by goal trees. For example, Fig- ure 3.16 shows a goal tree that can be used to represent the map-coloring problem for six countries with four colors. The tree has just two levels. The top level consists of a single and-node, which represents the fact that all countries must be colored. The next level has an or-node for each country, representing the choice of colors that can be applied.

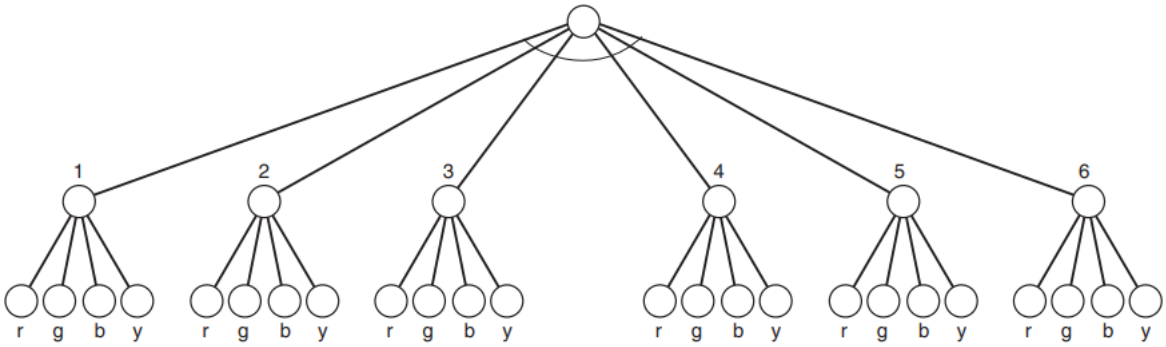


Figure 3.16
Goal tree representing a map-coloring problem with six countries and four colors

Of course, this tree alone does not represent the entire problem. **Constraints** must be applied that specify that no two adjacent countries may have the same color. Solving the tree while applying these constraints solves the map-coloring problem. In fact, to apply a search method to this problem, the goal tree must be redrawn as a search tree because search methods generally are not able to deal with and-nodes.

This can be done by redrawing the tree as a search tree, where paths through the tree represent **plans** rather than goals. Plans are discussed in more detail in Part 5 of this book. A plan consists of steps that can be taken to solve the overall problem. A search tree can thus be devised where nodes represent partial plans. The root node has no plan at all, and leaf nodes rep- resent complete plans.

A part of the search tree for the map-coloring problem with six countries and four colors is shown in Figure 3.17.

One of the search methods described in Chapter 4 or 5 can be applied to this search tree to find a solution. This may not be the most efficient way to solve the map-coloring problem, though.

Example 2: Proving Theorems

As will be explained in Part 3 of this book, goal trees can be used to represent theorems that are to be proved. The root goal is the theorem that is to

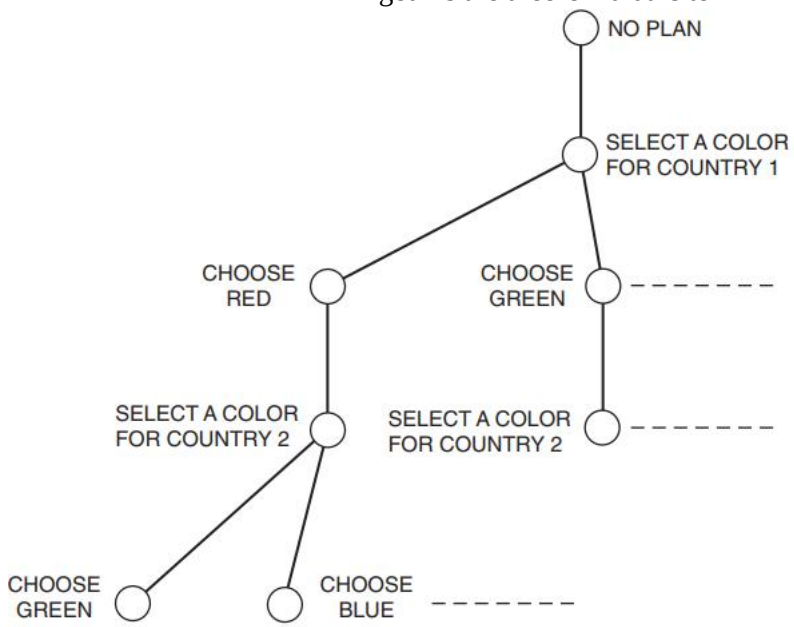


Figure 3.17
Partial search tree for map-coloring problem with six countries and four colors

be proved. It is an or-node because there may be several ways to prove the theorem. The next level down consists of and-nodes, which are lemmas that are to be proven. Each of these lemmas again may have several ways to be proved so, therefore, is an or-node. The leaf-nodes of the tree represent axioms that do not need to be proved.

Example 3: Parsing Sentences

As is described in Chapter 20, a parser is a tool that can be used to analyze the structure of a sentence in the English language (or any other human language). Sentences can be broken down into phrases, and phrases can be broken down into nouns, verbs, adjectives, and so on. Clearly, this is ideally suited to being represented by goal trees.

Example 4: Games

Game trees, which are described in more detail in Chapter 6, are goal trees that are used to represent the choices made by players when playing two-player games, such as chess, checkers, and Go. The root node of a game tree represents the current position, and this is an or-node because I must choose one move to make. The next level down in the game tree represents the possible choices my opponent might make, and because I need to consider all possible responses that I might make to that move, this level consists of and-nodes. Eventually, the leaf nodes represent final positions in the game, and a path through the tree represents a sequence of moves from start to finish, resulting in a win, loss, or a draw.

This kind of tree is a pure and-or tree because it has an or-node at the top, each or-node has and-nodes as its direct successors, and each and-node has or-nodes as its direct successors. Another condition of a pure and-or tree is that it does not have any constraints that affect which choices can be made.

Chapter Summary

- ✓ Artificial Intelligence can be used to solve a wide range of problems, but for the methods to work effectively, the correct representation must be used.
- ✓ Semantic nets use graphs to show relationships between objects. Frame-based systems show the same information in frames.
- ✓ Frame-based systems allow for inheritance, whereby one frame can inherit features from another.
- ✓ Frames often have procedures associated with them that enable a system to carry out actions on the basis of data within the frames.
- ✓ Search trees are a type of semantic tree. Search methods (several of which are described in Chapters 4 and 5) are applied to search trees, with the aim of finding a goal.
- ✓ Describe and Match is a method that can be used to identify an object by searching a tree that represents knowledge about the universe of objects that are being considered.
- ✓ Problems such as the Towers of Hanoi problem can be solved effectively by breaking them down into smaller subproblems, thus reducing an overall goal to a set of subgoals.
- ✓ Goal trees (or and-or trees) are an effective representation for problems that can be broken down in this way.
- ✓ Data-driven search (forward chaining) works from a start state toward a goal. Goal-driven search (backward chaining) works in the other direction, starting from the goal.