PART I

# ARTIFICIAL INTELLIGENCE: ITS ROOTS AND SCOPE

*Everything must have a beginning, to speak in Sanchean phrase; and that beginning must be linked to something that went before. Hindus give the world an elephant to support it, but they make the elephant stand upon a tortoise. Invention, it must be humbly admitted, does not consist in creating out of void, but out of chaos; the materials must, in the first place, be afforded. . . .*

—MARY SHELLEY, Frankenstein

## Artificial Intelligence: An Attempted Definition

*Artificial intelligence (AI) may be defined as the branch of computer science that is concerned with the automation of intelligent behavior.* This definition is particularly appropriate to this book in that it emphasizes our conviction that AI is a part of computer science and, as such, must be based on sound theoretical and applied principles of that field. These principles include the data structures used in knowledge representation, the algorithms needed to apply that knowledge, and the languages and programming techniques used in their implementation.

However, this definition suffers from the fact that intelligence itself is not very well defined or understood. Although most of us are certain that we know intelligent behavior when we see it, it is doubtful that anyone could come close to defining intelligence in a way that would be specific enough to help in the evaluation of a supposedly intelligent computer program, while still capturing the vitality and complexity of the human mind.

As a result of the daunting task of building a general intelligence, AI researchers often assume the roles of engineers fashioning particular intelligent artifacts. These often come in the form of diagnostic, prognostic, or visualization tools that enable their human users to perform complex tasks. Examples of these tools include hidden Markov models for language understanding, automated reasoning systems for proving new theorems in mathematics, dynammic Bayesian networks for tracking signals across cortical networks, and visualization of patterns of gene expression data, as seen in the applications of Section 1.2.

The problem of *defining* the full field of artificial intelligence becomes one of defining intelligence itself: is intelligence a single faculty, or is it just a name for a collection of distinct and unrelated abilities? To what extent is intelligence learned as opposed to having an a priori existence? Exactly what does happen when learning occurs? What is creativity? What is intuition? Can intelligence be inferred from observable behavior, or does it require evidence of a particular internal mechanism? How is knowledge represented in the nerve tissue of a living being, and what lessons does this have for the design of intelligent machines? What is self-awareness; what role does it play in intelligence? Furthermore, is it necessary to pattern an intelligent computer program after what is known about human intelligence, or is a strict "engineering" approach to the problem sufficient? Is it even possible to achieve intelligence on a computer, or does an intelligent entity require the richness of sensation and experience that might be found only in a biological existence?

These are unanswered questions, and all of them have helped to shape the problems and solution methodologies that constitute the core of modern AI. In fact, part of the appeal of artificial intelligence is that it offers a unique and powerful tool for exploring exactly these questions. AI offers a medium and a test-bed for theories of intelligence: such theories may be stated in the language of computer programs and consequently tested and verified through the execution of these programs on an actual computer.

For these reasons, our initial definition of artificial intelligence falls short of unambiguously defining the field. If anything, it has only led to further questions and the paradoxical notion of a field of study whose major goals include its own definition. But this difficulty in arriving at a precise definition of AI is entirely appropriate. Artificial intelligence is still a young discipline, and its structure, concerns, and methods are less clearly defined than those of a more mature science such as physics.

Artificial intelligence has always been more concerned with expanding the capabilities of computer science than with defining its limits. Keeping this exploration grounded in sound theoretical principles is one of the challenges facing AI researchers in general and this book in particular.

Because of its scope and ambition, artificial intelligence defies simple definition. For the time being, we will simply define it as *the collection of problems and methodologies studied by artificial intelligence researchers.* This definition may seem silly and meaningless, but it makes an important point: artificial intelligence, like every science, is a human endeavor, and perhaps, is best understood in that context.

There are reasons that any science, AI included, concerns itself with a certain set of problems and develops a particular body of techniques for approaching these problems. In Chapter 1, a short history of artificial intelligence and the people and assumptions that have shaped it will explain why certain sets of questions have come to dominate the field and why the methods discussed in this book have been taken for their solution.

# 2 THE PREDICATE CALCULUS

*We come to the full possession of our power of drawing inferences, the last of our faculties; for it is not so much a natural gift as a long and difficult art.*

—C. S. PIERCE

*The essential quality of a proof is to compel belief.*

—FERMAT

## 2.0 Introduction

In this chapter we introduce the predicate calculus as a representation language for artificial intelligence. The importance of the predicate calculus was discussed in the introduction to Part II; its advantages include a well-defined *formal semantics* and *sound* and *complete* inference rules. This chapter begins with a brief (optional) review of the propositional calculus (Section 2.1). Section 2.2 defines the syntax and semantics of the predicate calculus. In Section 2.3 we discuss predicate calculus inference rules and their use in problem solving. Finally, the chapter demonstrates the use of the predicate calculus to implement a knowledge base for financial investment advice.

## 2.1 The Propositional Calculus (optional)

### 2.1.1 Symbols and Sentences

The propositional calculus and, in the next subsection, the predicate calculus are first of all languages. Using their words, phrases, and sentences, we can represent and reason about properties and relationships in the world. The first step in describing a language is to introduce the pieces that make it up: its set of symbols.

DEFINITION

PROPOSITIONAL CALCULUS SYMBOLS

The *symbols* of propositional calculus are the propositional symbols:

P, Q, R, S, …

truth symbols:

true, false

and connectives:

$\wedge, \vee, \neg, \rightarrow, \equiv$

Propositional symbols denote *propositions*, or statements about the world that may be either true or false, such as "the car is red" or "water is wet." Propositions are denoted by uppercase letters near the end of the English alphabet. Sentences in the propositional calculus are formed from these atomic symbols according to the following rules:

DEFINITION

PROPOSITIONAL CALCULUS SENTENCES

Every propositional symbol and truth symbol is a sentence.

For example: true, P, Q, and R are sentences.

The *negation* of a sentence is a sentence.

For example: $\neg$ P and $\neg$ false are sentences.

The *conjunction*, or *and*, of two sentences is a sentence.

For example: $P \wedge \neg P$ is a sentence.

The *disjunction*, or *or*, of two sentences is a sentence.

For example: $P \vee \neg P$ is a sentence.

The *implication* of one sentence from another is a sentence.

For example: $P \rightarrow Q$ is a sentence.

The *equivalence* of two sentences is a sentence.

For example: $P \vee Q \equiv R$ is a sentence.

Legal sentences are also called *well-formed formulas* or *WFFs*.

In expressions of the form $P \wedge Q$, P and Q are called the *conjuncts*. In $P \vee Q$, P and Q are referred to as *disjuncts*. In an implication, $P \rightarrow Q$, P is the *premise* or *antecedent* and Q, the *conclusion* or *consequent*.

In propositional calculus sentences, the symbols ( ) and [ ] are used to group symbols into subexpressions and so to control their order of evaluation and meaning. For example, $(P \vee Q) \equiv R$ is quite different from $P \vee (Q \equiv R)$, as can be demonstrated using truth tables as we see Section 2.1.2.

An expression is a sentence, or well-formed formula, of the propositional calculus if and only if it can be formed of legal symbols through some sequence of these rules. For example,

$$((P \wedge Q) \rightarrow R) \equiv \neg P \vee \neg Q \vee R$$

is a well-formed sentence in the propositional calculus because:

P, Q, and R are propositions and thus sentences.

$P \wedge Q$, the conjunction of two sentences, is a sentence.

$(P \wedge Q) \rightarrow R$, the implication of a sentence for another, is a sentence.

$\neg P$ and $\neg Q$, the negations of sentences, are sentences.

$\neg P \vee \neg Q$, the disjunction of two sentences, is a sentence.

$\neg P \vee \neg Q \vee R$, the disjunction of two sentences, is a sentence.

$((P \wedge Q) \rightarrow R) \equiv \neg P \vee \neg Q \vee R$, the equivalence of two sentences, is a sentence.

This is our original sentence, which has been constructed through a series of applications of legal rules and is therefore “well formed”.

### 2.1.2 The Semantics of the Propositional Calculus

Section 2.1.1 presented the syntax of the propositional calculus by defining a set of rules for producing legal sentences. In this section we formally define the *semantics* or “meaning” of these sentences. Because AI programs must reason with their representational structures, it is important to demonstrate that the truth of their conclusions depends only on the truth of their initial knowledge or premises, i.e., that logical errors are not introduced by the inference procedures. A precise treatment of semantics is essential to this goal.

A proposition symbol corresponds to a statement about the world. For example, P may denote the statement “it is raining” or Q, the statement “I live in a brown house.” A proposition must be either true or false, given some state of the world. The truth value assignment to propositional sentences is called an *interpretation*, an assertion about their truth in some *possible world*.

Formally, an interpretation is a mapping from the propositional symbols into the set {T, F}. As mentioned in the previous section, the symbols true and false are part of the set of well-formed sentences of the propositional calculus; i.e., they are distinct from the truth value assigned to a sentence. To enforce this distinction, the symbols T and F are used for truth value assignment.

Each possible mapping of truth values onto propositions corresponds to a possible world of interpretation. For example, if P denotes the proposition "it is raining" and Q denotes "I am at work," then the set of propositions {P, Q} has four different functional mappings into the truth values {T, F}. These mappings correspond to four different interpretations. The semantics of propositional calculus, like its syntax, is defined inductively:

DEFINITION

PROPOSITIONAL CALCULUS SEMANTICS

An *interpretation* of a set of propositions is the assignment of a truth value, either T or F, to each propositional symbol.

The symbol true is always assigned T, and the symbol false is assigned F.

The interpretation or truth value for sentences is determined by:

The truth assignment of *negation*, $\neg$ P, where P is any propositional symbol, is F if the assignment to P is T, and T if the assignment to P is F.

The truth assignment of *conjunction*, $\wedge$, is T only when both conjuncts have truth value T; otherwise it is F.

The truth assignment of *disjunction*, $\vee$, is F only when both disjuncts have truth value F; otherwise it is T.

The truth assignment of *implication*, $\rightarrow$, is F only when the premise or symbol before the implication is T and the truth value of the consequent or symbol after the implication is F; otherwise it is T.

The truth assignment of *equivalence*, $\equiv$, is T only when both expressions have the same truth assignment for all possible interpretations; otherwise it is F.

The truth assignments of compound propositions are often described by *truth tables*. A truth table lists all possible truth value assignments to the atomic propositions of an expression and gives the truth value of the expression for each assignment. Thus, a truth table enumerates all possible worlds of interpretation that may be given to an expression. For example, the truth table for $P \wedge Q$, Figure 2.1, lists truth values for each possible truth assignment of the operands. $P \wedge Q$ is true only when P and Q are both T. Or ($\vee$), not ($\neg$), implies ($\rightarrow$), and equivalence ($\equiv$) are defined in a similar fashion. The construction of these truth tables is left as an exercise.

Two expressions in the propositional calculus are equivalent if they have the same value under all truth value assignments. This equivalence may be demonstrated using truth tables. For example, a proof of the equivalence of $P \rightarrow Q$ and $\neg P \vee Q$ is given by the truth table of Figure 2.2.

By demonstrating that two different sentences in the propositional calculus have identical truth tables, we can prove the following equivalences. For propositional expressions P, Q, and R:

$\neg (\neg P) \equiv P$

$(P \vee Q) \equiv (\neg P \rightarrow Q)$

the contrapositive law: $(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$

de Morgan's law: $\neg (P \vee Q) \equiv (\neg P \wedge \neg Q)$ and $\neg (P \wedge Q) \equiv (\neg P \vee \neg Q)$

the commutative laws: $(P \wedge Q) \equiv (Q \wedge P)$ and $(P \vee Q) \equiv (Q \vee P)$

the associative law: $((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$

the associative law: $((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$

the distributive law: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

the distributive law: $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$

Identities such as these can be used to change propositional calculus expressions into a syntactically different but logically equivalent form. These identities may be used instead of truth tables to prove that two expressions are equivalent: find a series of identities that transform one expression into the other. An early AI program, the *Logic Theorist* (Newell and Simon 1956), designed by Newell, Simon, and Shaw, used transformations between equivalent forms of expressions to prove many of the theorems in Whitehead and Russell's *Principia Mathematica* (1950). The ability to change a logical expression into a different form with equivalent truth values is also important when using inference rules (modus ponens, Section 2.3, and resolution, Chapter 14) that require expressions to be in a specific form.

| P | Q | P∧Q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

Figure 2.1 Truth table for the operator $\wedge$.

| P | Q | ¬P | ¬P∨Q | P⇒Q | (¬P∨Q)=(P⇒Q) |
|---|---|---|---|---|---|
| T | T | F | T | T | T |
| T | F | F | F | F | T |
| F | T | T | T | T | T |
| F | F | T | T | T | T |

Figure 2.2 Truth table demonstrating the equivalence of $P \rightarrow Q$ and $\neg P \vee Q$.

## 2.2 The Predicate Calculus

In propositional calculus, each atomic symbol (P, Q, etc.) denotes a single proposition. There is no way to access the components of an individual assertion. Predicate calculus provides this ability. For example, instead of letting a single propositional symbol, P, denote the entire sentence "it rained on Tuesday," we can create a predicate weather that describes a relationship between a date and the weather: weather(tuesday, rain). Through inference rules we can manipulate predicate calculus expressions, accessing their individual components and inferring new sentences.

Predicate calculus also allows expressions to contain variables. Variables let us create general assertions about classes of entities. For example, we could state that for all values of X, where X is a day of the week, the statement weather(X, rain) is true; i.e., it rains every day. As we did with the propositional calculus, we will first define the syntax of the language and then discuss its semantics.

### 2.2.1 The Syntax of Predicates and Sentences

Before defining the syntax of correct expressions in the predicate calculus, we define an alphabet and grammar for creating the *symbols* of the language. This corresponds to the lexical aspect of a programming language definition. Predicate calculus symbols, like the *tokens* in a programming language, are irreducible syntactic elements: they cannot be broken into their component parts by the operations of the language.

In our presentation we represent predicate calculus symbols as strings of letters and digits beginning with a letter. Blanks and nonalphanumeric characters cannot appear within the string, although the underscore, _, may be used to improve readability.

DEFINITION

PREDICATE CALCULUS SYMBOLS

The alphabet that makes up the symbols of the predicate calculus consists of:

1. The set of letters, both upper- and lowercase, of the English alphabet.
2. The set of digits, 0, 1, …, 9.
3. The underscore, _.

*Symbols* in the predicate calculus begin with a letter and are followed by any sequence of these legal characters.

Legitimate characters in the alphabet of predicate calculus symbols include

a R 6 9 p _ z

Examples of characters not in the alphabet include

# % @ / &

Legitimate predicate calculus symbols include

George   fire3   tom_and_jerry   bill   XXXX   friends_of

Examples of strings that are not legal symbols are

3jack   no blanks allowed   ab%cd   ***71   duck!!!

Symbols, as we see in Section 2.2.2, are used to denote objects, properties, or relations in a world of discourse. As with most programming languages, the use of "words" that suggest the symbol's intended meaning assists us in understanding program code. Thus, even though l(g,k) and likes(george, kate) are formally equivalent (i.e., they have the same structure), the second can be of great help (for human readers) in indicating what relationship the expression represents. It must be stressed that these descriptive names are intended solely to improve the readability of expressions. The only "meaning" that predicate calculus expressions have is given through their formal semantics.

Parentheses "( )", commas ",", and periods "." are used solely to construct well-formed expressions and do not denote objects or relations in the world. These are called *improper symbols*.

Predicate calculus symbols may represent either *variables, constants, functions*, or *predicates*. Constants name specific objects or properties in the world. Constant symbols must begin with a lowercase letter. Thus george, tree, tall, and blue are examples of well-formed constant symbols. The constants true and false are reserved as *truth symbols*.

Variable symbols are used to designate general classes of objects or properties in the world. Variables are represented by symbols beginning with an uppercase letter. Thus George, BILL, and KAte are legal variables, whereas geORGE and bill are not.

Predicate calculus also allows functions on objects in the world of discourse. Function symbols (like constants) begin with a lowercase letter. Functions denote a mapping of one or more elements in a set (called the *domain* of the function) into a unique element of a second set (the *range* of the function). Elements of the domain and range are objects in the world of discourse. In addition to common arithmetic functions such as addition and multiplication, functions may define mappings between nonnumeric domains.

Note that our definition of predicate calculus symbols does not include numbers or arithmetic operators. The number system is not included in the predicate calculus primitives; instead it is defined axiomatically using "pure" predicate calculus as a basis (Manna and Waldinger 1985). While the particulars of this derivation are of theoretical interest, they are less important to the use of predicate calculus as an AI representation language. For convenience, we assume this derivation and include arithmetic in the language.

Every function symbol has an associated *arity*, indicating the number of elements in the domain mapped onto each element of the range. Thus father could denote a function of arity 1 that maps people onto their (unique) male parent. plus could be a function of arity 2 that maps two numbers onto their arithmetic sum.

A *function expression* is a function symbol followed by its arguments. The arguments are elements from the domain of the function; the number of arguments is equal to the

arity of the function. The arguments are enclosed in parentheses and separated by commas. For example,

```
f(X,Y)
father(david)
price(bananas)
```

are all well-formed function expressions.

Each function expression denotes the mapping of the arguments onto a single object in the range, called the *value* of the function. For example, if father is a unary function, then

```
father(david)
```

is a function expression whose value (in the author's world of discourse) is george. If plus is a function of arity 2, with domain the integers, then

```
plus(2,3)
```

is a function expression whose value is the integer 5. The act of replacing a function with its value is called *evaluation*.

The concept of a predicate calculus symbol or term is formalized in the following definition:

DEFINITION

SYMBOLS and TERMS

Predicate calculus symbols include:

1. *Truth symbols* true and false (these are reserved symbols).
2. *Constant symbols* are symbol expressions having the first character lowercase.
3. *Variable symbols* are symbol expressions beginning with an uppercase character.
4. *Function symbols* are symbol expressions having the first character lowercase. Functions have an attached arity indicating the number of elements of the domain mapped onto each element of the range.

A *function expression* consists of a function constant of arity $n$, followed by $n$ terms, $t_1, t_2, \cdots, t_n$, enclosed in parentheses and separated by commas.

A predicate calculus *term* is either a constant, variable, or function expression.

Thus, a predicate calculus *term* may be used to denote objects and properties in a problem domain. Examples of terms are:

cat
times(2,3)
X
blue
mother(sarah)
kate

Symbols in predicate calculus may also represent predicates. Predicate symbols, like constants and function names, begin with a lowercase letter. A predicate names a relationship between zero or more objects in the world. The number of objects so related is the arity of the predicate. Examples of predicates are

likes equals on near part_of

An *atomic sentence*, the most primitive unit of the predicate calculus language, is a predicate of arity n followed by n terms enclosed in parentheses and separated by commas. Examples of atomic sentences are

| | |
|---|---|
| likes(george,kate) | likes(X,george) |
| likes(george,susie) | likes(X,X) |
| likes(george,sarah,tuesday) | friends(bill,richard) |
| friends(bill,george) | friends(father_of(david),father_of(andrew)) |
| helps(bill,george) | helps(richard,bill) |

The predicate symbols in these expressions are likes, friends, and helps. A predicate symbol may be used with different numbers of arguments. In this example there are two different likes, one with two and the other with three arguments. When a predicate symbol is used in sentences with different arities, it is considered to represent two different relations. Thus, a predicate relation is defined by its name and its arity. There is no reason that the two different likes cannot make up part of the same description of the world; however, this is usually avoided because it can often cause confusion.

In the predicates above, bill, george, kate, etc., are constant symbols and represent objects in the problem domain. The arguments to a predicate are terms and may also include variables or function expressions. For example,

friends(father_of(david),father_of(andrew))

is a predicate describing a relationship between two objects in a domain of discourse. These arguments are represented as function expressions whose mappings (given that the father_of david is george and the father_of andrew is allen) form the parameters of the predicate. If the function expressions are evaluated, the expression becomes

friends(george,allen)

These ideas are formalized in the following definition.

DEFINITION

PREDICATES and ATOMIC SENTENCES

Predicate symbols are symbols beginning with a lowercase letter.

Predicates have an associated positive integer referred to as the *arity* or “argument number” for the predicate. Predicates with the same name but different arities are considered distinct.

An atomic sentence is a predicate constant of arity n, followed by n terms, $t_1, t_2, \cdots, t_n$, enclosed in parentheses and separated by commas.

The truth values, true and false, are also atomic sentences.

Atomic sentences are also called *atomic expressions*, *atoms*, or *propositions*.

We may combine atomic sentences using logical operators to form *sentences* in the predicate calculus. These are the same logical connectives used in propositional calculus: $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\equiv$.

When a variable appears as an argument in a sentence, it refers to unspecified objects in the domain. First order (Section 2.2.2) predicate calculus includes two symbols, the *variable quantifiers* $\forall$ and $\exists$, that constrain the meaning of a sentence containing a variable. A quantifier is followed by a variable and a sentence, such as

∃ Y friends(Y, peter)
∀ X likes(X, ice_cream)

The *universal quantifier*, $\forall$, indicates that the sentence is true for all values of the variable. In the example, ∀ X likes(X, ice_cream) is true for all values in the domain of the definition of X. The *existential quantifier*, $\exists$, indicates that the sentence is true for at least one value in the domain. ∃ Y friends(Y, peter) is true if there is at least one object, indicated by Y that is a friend of peter. Quantifiers are discussed in more detail in Section 2.2.2.

Sentences in the predicate calculus are defined inductively.

DEFINITION

PREDICATE CALCULUS SENTENCES

Every atomic sentence is a sentence.

1. If s is a sentence, then so is its negation, $\neg$ s.
2. If $s_1$ and $s_2$ are sentences, then so is their conjunction, $s_1 \wedge s_2$.
3. If $s_1$ and $s_2$ are sentences, then so is their disjunction, $s_1 \vee s_2$.
4. If $s_1$ and $s_2$ are sentences, then so is their implication, $s_1 \rightarrow s_2$.
5. If $s_1$ and $s_2$ are sentences, then so is their equivalence, $s_1 \equiv s_2$.

6. If X is a variable and s a sentence, then ∀ X s is a sentence.

7. If X is a variable and s a sentence, then ∃ X s is a sentence.

Examples of well-formed sentences follow. Let times and plus be function symbols of arity 2 and let equal and foo be predicate symbols with arity 2 and 3, respectively.

plus(two,three) is a function and thus not an atomic sentence.

equal(plus(two,three), five) is an atomic sentence.

equal(plus(2, 3), seven) is an atomic sentence. Note that this sentence, given the standard interpretation of plus and equal, is false. Well-formedness and truth value are independent issues.

∃ X foo(X,two,plus(two,three)) ∧ equal(plus(two,three),five) is a sentence since both conjuncts are sentences.

(foo(two,two,plus(two,three))) → (equal(plus(three,two),five) ≡ true) is a sentence because all its components are sentences, appropriately connected by logical operators.

The definition of predicate calculus sentences and the examples just presented suggest a method for verifying that an expression is a sentence. This is written as a recursive algorithm, verify_sentence. verify_sentence takes as argument a candidate expression and returns success if the expression is a sentence.

```
function verify_sentence(expression);
begin
  case
    expression is an atomic sentence: return SUCCESS;
    expression is of the form Q X s, where Q is either ∀ or ∃, X is a variable,
      if verify_sentence(s) returns SUCCESS
      then return SUCCESS
      else return FAIL;
    expression is of the form ¬ s:
      if verify_sentence(s) returns SUCCESS
      then return SUCCESS
      else return FAIL;
    expression is of the form s1 op s2, where op is a binary logical operator:
      if verify_sentence(s1) returns SUCCESS and
        verify_sentence(s2) returns SUCCESS
      then return SUCCESS
      else return FAIL;
    otherwise: return FAIL
  end
end.
```

We conclude this section with an example of the use of predicate calculus to describe a simple world. The domain of discourse is a set of family relationships in a biblical genealogy:

mother(eve,abel)
mother(eve,cain)
father(adam,abel)
father(adam,cain)

$\forall$ X $\forall$ Y father(X, Y) $\vee$ mother(X, Y) $\rightarrow$ parent(X, Y)
$\forall$ X $\forall$ Y $\forall$ Z parent(X, Y) $\wedge$ parent(X, Z) $\rightarrow$ sibling(Y, Z)

In this example we use the predicates mother and father to define a set of parent–child relationships. The implications give general definitions of other relationships, such as parent and sibling, in terms of these predicates. Intuitively, it is clear that these implications can be used to infer facts such as sibling(cain,abel). To formalize this process so that it can be performed on a computer, care must be taken to define inference algorithms and to ensure that such algorithms indeed draw correct conclusions from a set of predicate calculus assertions. In order to do so, we define the semantics of the predicate calculus (Section 2.2.2) and then address the issue of inference rules (Section 2.3).

### 2.2.2 A Semantics for the Predicate Calculus

Having defined well-formed expressions in the predicate calculus, it is important to determine their meaning in terms of objects, properties, and relations in the world. Predicate calculus semantics provide a formal basis for determining the truth value of well-formed expressions. The truth of expressions depends on the mapping of constants, variables, predicates, and functions into objects and relations in the domain of discourse. The truth of relationships in the domain determines the truth of the corresponding expressions.

For example, information about a person, George, and his friends Kate and Susie may be expressed by

friends(george,susie)
friends(george,kate)

If it is indeed true that George is a friend of Susie and George is a friend of Kate then these expressions would each have the truth value (assignment) T. If George is a friend of Susie but not of Kate, then the first expression would have truth value T and the second would have truth value F.

To use the predicate calculus as a representation for problem solving, we describe objects and relations in the domain of interpretation with a set of well-formed expressions. The terms and predicates of these expressions denote objects and relations in the domain. This database of predicate calculus expressions, each having truth value T, describes the

"state of the world". The description of George and his friends is a simple example of such a database. Another example is the *blocks world* in the introduction to Part II.

Based on these intuitions, we formally define the semantics of predicate calculus. First, we define an *interpretation* over a domain D. Then we use this interpretation to determine the *truth value assignment* of sentences in the language.

DEFINITION

INTERPRETATION

Let the domain D be a nonempty set.

An *interpretation* over D is an assignment of the entities of D to each of the constant, variable, predicate, and function symbols of a predicate calculus expression, such that:

1. Each constant is assigned an element of D.
2. Each variable is assigned to a nonempty subset of D; these are the allowable substitutions for that variable.
3. Each function f of arity m is defined on m arguments of D and defines a mapping from $D^m$ into D.
4. Each predicate p of arity n is defined on n arguments from D and defines a mapping from $D^n$ into {T, F}.

Given an interpretation, the meaning of an expression is a truth value assignment over the interpretation.

DEFINITION

TRUTH VALUE OF PREDICATE CALCULUS EXPRESSIONS

Assume an expression E and an interpretation I for E over a nonempty domain D. The truth value for E is determined by:

1. The value of a constant is the element of D it is assigned to by I.
2. The value of a variable is the set of elements of D it is assigned to by I.
3. The value of a function expression is that element of D obtained by evaluating the function for the parameter values assigned by the interpretation.
4. The value of truth symbol "true" is T and "false" is F.
5. The value of an atomic sentence is either T or F, as determined by the interpretation I.

6. The value of the negation of a sentence is T if the value of the sentence is F and is F if the value of the sentence is T.

7. The value of the conjunction of two sentences is T if the value of both sentences is T and is F otherwise.

8.–10. The truth value of expressions using ∨, →, and ≡ is determined from the value of their operands as defined in Section 2.1.2.

Finally, for a variable X and a sentence S containing X:

11. The value of ∀ X S is T if S is T for all assignments to X under I, and it is F otherwise.

12. The value of ∃ X S is T if there is an assignment to X in the interpretation under which S is T; otherwise it is F.

Quantification of variables is an important part of predicate calculus semantics. When a variable appears in a sentence, such as X in likes(george,X), the variable functions as a placeholder. Any constant allowed under the interpretation can be substituted for it in the expression. Substituting kate or susie for X in likes(george,X) forms the statements likes(george,kate) and likes(george,susie) as we saw earlier.

The variable X stands for all constants that might appear as the second parameter of the sentence. This variable name might be replaced by any other variable name, such as Y or PEOPLE, without changing the meaning of the expression. Thus the variable is said to be a *dummy.* In the predicate calculus, variables must be *quantified* in either of two ways: *universally* or *existentially.* A variable is considered *free* if it is not within the scope of either the universal or existential quantifiers. An expression is *closed* if all of its variables are quantified. A *ground expression* has no variables at all. In the predicate calculus all variables must be quantified.

Parentheses are often used to indicate the *scope* of quantification, that is, the instances of a variable name over which a quantification holds. Thus, for the symbol indicating universal quantification, ∀:

∀ X (p(X) ∨ q(Y) → r(X))

indicates that X is universally quantified in both p(X) and r(X).

Universal quantification introduces problems in computing the truth value of a sentence, because all the possible values of a variable symbol must be tested to see whether the expression remains true. For example, to test the truth value of ∀ X likes(george,X), where X ranges over the set of all humans, all possible values for X must be tested. If the domain of an interpretation is infinite, exhaustive testing of all substitutions to a universally quantified variable is computationally impossible: the algorithm may never halt. Because of this problem, the predicate calculus is said to be *undecidable*. Because the propositional calculus does not support variables, sentences can only have a finite number of truth assignments, and we can exhaustively test all these possible assignments. This is done with the truth table, Section 2.1.

As seen in Section 2.2.1 variables may also be quantified *existentially,* indicated by the symbol $\exists$. In the existential case the expression containing the variable is said to be true for at least one substitution from its domain of definition. The scope of an existentially quantified variable is also indicated by enclosing the quantified occurrences of the variable in parentheses.

Evaluating the truth of an expression containing an existentially quantified variable may be no easier than evaluating the truth of expressions containing universally quantified variables. Suppose we attempt to determine the truth of the expression by trying substitutions until one is found that makes the expression true. If the domain of the variable is infinite and the expression is false under all substitutions, the algorithm will never halt.

Several relationships between negation and the universal and existential quantifiers are given below. These relationships are used in resolution refutation systems described in Chapter 14. The notion of a variable name as a dummy symbol that stands for a set of constants is also noted. For predicates p and q and variables X and Y:

$$\neg \exists X\ p(X) \equiv \forall X \neg p(X)$$

$$\neg \forall X\ p(X) \equiv \exists X \neg p(X)$$

$$\exists X\ p(X) \equiv \exists Y\ p(Y)$$

$$\forall X\ q(X) \equiv \forall Y\ q(Y)$$

$$\forall X\ (p(X) \wedge q(X)) \equiv \forall X\ p(X) \wedge \forall Y\ q(Y)$$

$$\exists X\ (p(X) \vee q(X)) \equiv \exists X\ p(X) \vee \exists Y\ q(Y)$$

In the language we have defined, universally and existentially quantified variables may refer only to objects (constants) in the domain of discourse. Predicate and function names may not be replaced by quantified variables. This language is called the *first-order predicate calculus*.

DEFINITION

FIRST-ORDER PREDICATE CALCULUS

*First-order predicate calculus* allows quantified variables to refer to objects in the domain of discourse and not to predicates or functions.

For example,

$$\forall\ (Likes)\ Likes(george,kate)$$

is not a well-formed expression in the first-order predicate calculus. There are *higher-order* predicate calculi where such expressions are meaningful. Some researchers

(McCarthy 1968, Appelt 1985) have used higher-order languages to represent knowledge in natural language understanding programs.

Many grammatically correct English sentences can be represented in the first-order predicate calculus using the symbols, connectives, and variable symbols defined in this section. It is important to note that there is no unique mapping of sentences into predicate calculus expressions; in fact, an English sentence may have any number of different predicate calculus representations. A major challenge for AI programmers is to find a scheme for using these predicates that optimizes the expressiveness and efficiency of the resulting representation. Examples of English sentences represented in predicate calculus are:

If it doesn't rain on Monday, Tom will go to the mountains.
$\neg$ weather(rain, monday) $\rightarrow$ go(tom, mountains)

Emma is a Doberman pinscher and a good dog.
gooddog(emma) $\wedge$ isa(emma, doberman)

All basketball players are tall.
$\forall$ X (basketball_player(X) $\rightarrow$ tall(X))

Some people like anchovies.
$\exists$ X (person(X) $\wedge$ likes(X, anchovies))

If wishes were horses, beggars would ride.
equal(wishes, horses) $\rightarrow$ ride(beggars)

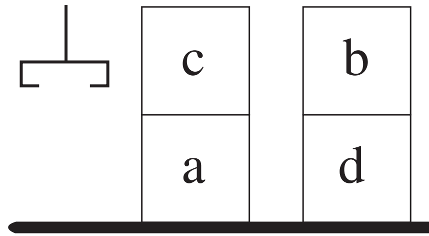
Nobody likes taxes.
$\neg$ $\exists$ X likes(X, taxes)

### 2.2.3 A "Blocks World" Example of Semantic Meaning

We conclude this section by giving an extended example of a truth value assignment to a set of predicate calculus expressions. Suppose we want to model the blocks world of Figure 2.3 to design, for example, a control algorithm for a robot arm. We can use predicate calculus sentences to represent the qualitative relationships in the world: does a given block have a clear top surface? can we pick up block a? etc. Assume that the computer has knowledge of the location of each block and the arm and is able to keep track of these locations (using three-dimensional coordinates) as the hand moves blocks about the table.

We must be very precise about what we are proposing with this "blocks world" example. First, we are creating a set of predicate calculus expressions that is to represent a static snapshot of the blocks world problem domain. As we will see in Section 2.3, this set of blocks offers an *interpretation* and a possible *model* for the set of predicate calculus expressions.

Second, the predicate calculus is *declarative*, that is, there is no assumed timing or order for considering each expression. Nonetheless, in the planning section of this book, Section 8.4, we will add a "procedural semantics", or a clearly specified methodology for evaluating these expressions over time. A concrete example of a procedural semantics for predicate calculus expressions is Prolog, Section 14.3. This situation calculus we are




on(c,a)
on(b,d)
ontable(a)
ontable(d)
clear(b)
clear(c)
hand_empty

Figure 2.3 A blocks world with its predicate calculus description.

creating will introduce a number of issues, including the *frame problem* and the issue of *non-monotonicity* of logic interpretations, that will be addressed later in this book. For this example, however, it is sufficient to say that our predicate calculus expressions will be evaluated in a top-down and left-to-right fashion.

To pick up a block and stack it on another block, both blocks must be clear. In Figure 2.3, block a is not clear. Because the arm can move blocks, it can change the state of the world and clear a block. Suppose it removes block c from block a and updates the knowledge base to reflect this by deleting the assertion on(c,a). The program needs to be able to infer that block a has become clear.

The following rule describes when a block is clear:

$$\forall X (\neg \exists Y \ on(Y,X) \rightarrow clear(X))$$

That is, for all X, X is clear if there does not exist a Y such that Y is on X.

This rule not only defines what it means for a block to be clear but also provides a basis for determining how to clear blocks that are not. For example, block d is not clear, because if variable X is given value d, substituting b for Y will make the statement false. Therefore, to make this definition true, block b must be removed from block d. This is easily done because the computer has a record of all the blocks and their locations.

Besides using implications to define when a block is clear, other rules may be added that describe operations such as stacking one block on top of another. For example: to stack X on Y, first empty the hand, then clear X, then clear Y, and then pick_up X and put_down X on Y.

$$\forall X \forall Y ((hand_empty \land clear(X) \land clear(Y) \land pick_up(X) \land put_down(X,Y)) \rightarrow stack(X,Y))$$

Note that in implementing the above description it is necessary to "attach" an action of the robot arm to each predicate such as pick_up(X). As noted previously, for such an implementation it was necessary to augment the semantics of predicate calculus by requiring that the actions be performed in the order in which they appear in a rule premise. However, much is gained by separating these issues from the use of predicate calculus to define the relationships and operations in the domain.

Figure 2.3 gives a semantic interpretation of these predicate calculus expressions. This interpretation maps the constants and predicates in the set of expressions into a domain D, here the blocks and relations between them. The interpretation gives truth value T to each expression in the description. Another interpretation could be offered by a different set of blocks in another location, or perhaps by a team of four acrobats. The important question is not the uniqueness of interpretations, but whether the interpretation provides a truth value for all expressions in the set and whether the expressions describe the world in sufficient detail that all necessary inferences may be carried out by manipulating the symbolic expressions. The next section uses these ideas to provide a formal basis for predicate calculus inference rules.

## 2.3 Using Inference Rules to Produce Predicate Calculus Expressions

### 2.3.1 Inference Rules

The semantics of the predicate calculus provides a basis for a formal theory of *logical inference*. The ability to infer new correct expressions from a set of true assertions is an important feature of the predicate calculus. These new expressions are correct in that they are *consistent* with all previous interpretations of the original set of expressions. First we discuss these ideas informally and then we create a set of definitions to make them precise.

An interpretation that makes a sentence true is said to *satisfy* that sentence. An interpretation that satisfies every member of a set of expressions is said to satisfy the set. An expression X *logically follows* from a set of predicate calculus expressions S if every interpretation that satisfies S also satisfies X. This notion gives us a basis for verifying the correctness of rules of inference: the function of logical inference is to produce new sentences that logically follow a given set of predicate calculus sentences.

It is important that the precise meaning of *logically follows* be understood: for expression X to *logically follow* S, it must be true for every interpretation that satisfies the original set of expressions S. This would mean, for example, that any new predicate calculus expression added to the blocks world of Figure 2.3 must be true in that world as well as in any other interpretation that that set of expressions may have.

The term itself, "logically follows," may be a bit confusing. It does not mean that X is deduced from or even that it is deducible from S. It simply means that X is true for every

interpretation that satisfies S. However, because systems of predicates can have a potentially infinite number of possible interpretations, it is seldom practical to try all interpretations. Instead, *inference rules* provide a computationally feasible way to determine when an expression, a component of an interpretation, logically follows for that interpretation. The concept "logically follows" provides a formal basis for proofs of the soundness and correctness of inference rules.

An inference rule is essentially a mechanical means of producing new predicate calculus sentences from other sentences. That is, inference rules produce new sentences based on the syntactic form of given logical assertions. When every sentence X produced by an inference rule operating on a set S of logical expressions logically follows from S, the inference rule is said to be *sound*.

If the inference rule is able to produce every expression that logically follows from S, then it is said to be *complete*. *Modus ponens*, to be introduced below, and *resolution*, introduced in Chapter 11, are examples of inference rules that are sound and, when used with appropriate application strategies, complete. Logical inference systems generally use sound rules of inference, although later chapters (4, 9, and 15) examine heuristic reasoning and commonsense reasoning, both of which relax this requirement.

We formalize these ideas through the following definitions.

DEFINITION

SATISFY, MODEL, VALID, INCONSISTENT

For a predicate calculus expression X and an interpretation I:

If X has a value of T under I and a particular variable assignment, then I is said to *satisfy* X.

If I satisfies X for all variable assignments, then I is a *model* of X.

X is *satisfiable* if and only if there exist an interpretation and variable assignment that satisfy it; otherwise, it is *unsatisfiable*.

A set of expressions is *satisfiable* if and only if there exist an interpretation and variable assignment that satisfy every element.

If a set of expressions is not satisfiable, it is said to be *inconsistent*.

If X has a value T for all possible interpretations, X is said to be *valid*.

In the blocks world example of Figure 2.3, the blocks world was a model for its logical description. All of the sentences in the example were true under this interpretation. When a knowledge base is implemented as a set of true assertions about a problem domain, that domain is a model for the knowledge base.

The expression $\exists$ X (p(X) $\wedge \neg$ p(X)) is inconsistent, because it cannot be satisfied under any interpretation or variable assignment. On the other hand, the expression $\forall$ X (p(X) $\vee \neg$ p(X)) is valid.

The truth table method can be used to test validity for any expression not containing variables. Because it is not always possible to decide the validity of expressions containing variables (as mentioned above, the process may not terminate), the full predicate calculus is “undecidable.” There are *proof procedures*, however, that can produce any expression that logically follows from a set of expressions. These are called *complete* proof procedures.

DEFINITION

PROOF PROCEDURE

A *proof procedure* is a combination of an inference rule and an algorithm for applying that rule to a set of logical expressions to generate new sentences.

We present proof procedures for the *resolution* inference rule in Chapter 11.

Using these definitions, we may formally define “logically follows.”

DEFINITION

LOGICALLY FOLLOWS, SOUND, and COMPLETE

A predicate calculus expression X *logically follows* from a set S of predicate calculus expressions if every interpretation and variable assignment that satisfies S also satisfies X.

An inference rule is *sound* if every predicate calculus expression produced by the rule from a set S of predicate calculus expressions also logically follows from S.

An inference rule is *complete* if, given a set S of predicate calculus expressions, the rule can infer every expression that logically follows from S.

Modus ponens is a sound inference rule. If we are given an expression of the form $P \rightarrow Q$ and another expression of the form P such that both are true under an interpretation I, then modus ponens allows us to infer that Q is also true for that interpretation. Indeed, because modus ponens is sound, Q is true for *all* interpretations for which P and $P \rightarrow Q$ are true.

Modus ponens and a number of other useful inference rules are defined below.

DEFINITION

MODUS PONENS, MODUS TOLLENS, AND ELIMINATION, AND INTRODUCTION, and UNIVERSAL INSTANTIATION

If the sentences P and $P \rightarrow Q$ are known to be true, then *modus ponens* lets us infer Q.

Under the inference rule *modus tollens*, if $P \rightarrow Q$ is known to be true and Q is known to be false, we can infer that P is false: $\neg P$.

*And elimination* allows us to infer the truth of either of the conjuncts from the truth of a conjunctive sentence. For instance, $P \wedge Q$ lets us conclude P and Q are true.

*And introduction* lets us infer the truth of a conjunction from the truth of its conjuncts. For instance, if P and Q are true, then $P \wedge Q$ is true.

*Universal instantiation* states that if any universally quantified variable in a true sentence, say p(X), is replaced by an appropriate term from the domain, the result is a true sentence. Thus, if a is from the domain of X, $\forall X\ p(X)$ lets us infer p(a).

As a simple example of the use of modus ponens in the propositional calculus, assume the following observations: "if it is raining then the ground is wet" and "it is raining." If P denotes "it is raining" and Q is "the ground is wet" then the first expression becomes $P \rightarrow Q$. Because it is indeed now raining (P is true), our set of axioms becomes

$P \rightarrow Q$
P

Through an application of modus ponens, the fact that the ground is wet (Q) may be added to the set of true expressions.

Modus ponens can also be applied to expressions containing variables. Consider as an example the common syllogism "all men are mortal and Socrates is a man; therefore Socrates is mortal." "All men are mortal" may be represented in predicate calculus by:

$\forall X\ (man(X) \rightarrow mortal(X))$.

"Socrates is a man" is

man(socrates).

Because the X in the implication is universally quantified, we may substitute any value in the domain for X and still have a true statement under the inference rule of universal instantiation. By substituting socrates for X in the implication, we infer the expression

$man(socrates) \rightarrow mortal(socrates)$.

We can now apply modus ponens and infer the conclusion mortal(socrates). This is added to the set of expressions that logically follow from the original assertions. An algorithm called *unification* can be used by an automated problem solver to determine that socrates may be substituted for X in order to apply modus ponens. Unification is discussed in Section 2.3.2.

Chapter 14 discusses a more powerful rule of inference called *resolution*, which is the basis of many automated reasoning systems.

### 2.3.2 Unification

To apply inference rules such as modus ponens, an inference system must be able to determine when two expressions are the same or *match*. In propositional calculus, this is trivial: two expressions match if and only if they are syntactically identical. In predicate calculus, the process of matching two sentences is complicated by the existence of variables in the expressions. Universal instantiation allows universally quantified variables to be replaced by terms from the domain. This requires a decision process for determining the variable substitutions under which two or more expressions can be made identical (usually for the purpose of applying inference rules).

*Unification* is an algorithm for determining the substitutions needed to make two predicate calculus expressions match. We have already seen this done in the previous subsection, where socrates in man(socrates) was substituted for X in ∀ X(man(X) ⇒ mortal(X)). This allowed the application of modus ponens and the conclusion mortal(socrates). Another example of unification was seen previously when dummy variables were discussed. Because p(X) and p(Y) are equivalent, Y may be substituted for X to make the sentences match.

Unification and inference rules such as modus ponens allow us to make inferences on a set of logical assertions. To do this, the logical database must be expressed in an appropriate form.

An essential aspect of this form is the requirement that all variables be universally quantified. This allows full freedom in computing substitutions. Existentially quantified variables may be eliminated from sentences in the database by replacing them with the constants that make the sentence true. For example, ∃ X parent(X,tom) could be replaced by the expression parent(bob,tom) or parent(mary,tom), assuming that bob and mary are tom s parents under the interpretation.

The process of eliminating existentially quantified variables is complicated by the fact that the value of these substitutions may depend on the value of other variables in the expression. For example, in the expression ∀ X ∃ Y mother(X,Y), the value of the existentially quantified variable Y depends on the value of X. *Skolemization* replaces each existentially quantified variable with a function that returns the appropriate constant as a function of some or all of the other variables in the sentence. In the above example, because the value of Y depends on X, Y could be replaced by a *skolem function*, f, of X. This yields the predicate ∀ X mother(X,f(X)). Skolemization, a process that can also bind universally quantified variables to constants, is discussed in more detail in Section 14.2.

Once the existentially quantified variables have been removed from a logical database, unification may be used to match sentences in order to apply inference rules such as modus ponens.

Unification is complicated by the fact that a variable may be replaced by any term, including other variables and function expressions of arbitrary complexity. These expressions may themselves contain variables. For example, father(jack) may be substituted for X in man(X) to infer that jack s father is mortal.

Some instances of the expression

foo(X,a,goo(Y)).

generated by legal substitutions are given below:

1) foo(fred,a,goo(Z))
2) foo(W,a,goo(jack))
3) foo(Z,a,goo(moo(Z)))

In this example, the substitution instances or *unifications* that would make the initial expression identical to each of the other three are written as the sets:

1) {fred/X, Z/Y}
2) {W/X, jack/Y}
3) {Z/X, moo(Z)/Y}

The notation X/Y,… indicates that X is substituted for the variable Y in the original expression. Substitutions are also referred to as *bindings*. A variable is said to be *bound* to the value substituted for it.

In defining the unification algorithm that computes the substitutions required to match two expressions, a number of issues must be taken into account. First, although a constant may be systematically substituted for a variable, any constant is considered a "ground instance" and may not be replaced. Neither can two different ground instances be substituted for one variable. Second, a variable cannot be unified with a term containing that variable. X cannot be replaced by p(X) as this creates an infinite expression: p(p(p(p(...X)...))). The test for this situation is called the *occurs check*.

Furthermore, a problem-solving process often requires multiple inferences and, consequently, multiple successive unifications. Logic problem solvers must maintain consistency of variable substitutions. It is important that any unifying substitution be made consistently across all occurrences within the scope of the variable in both expressions being matched. This was seen before when socrates was substituted not only for the variable X in man(X) but also for the variable X in mortal(X).

Once a variable has been bound, future unifications and inferences must take the value of this binding into account. If a variable is bound to a constant, that variable may not be given a new binding in a future unification. If a variable $X_1$ is substituted for another variable $X_2$ and at a later time $X_1$ is replaced by a constant, then $X_2$ must also reflect this binding. The set of substitutions used in a sequence of inferences is important, because it may contain the answer to the original query (Section 14.2.5). For example, if p(a,X) unifies with the premise of p(Y,Z) $\Rightarrow$ q(Y,Z) with substitution {a/Y, X/Z}, modus ponens lets us infer q(a,X) under the same substitution. If we match this result with the premise of q(W,b) $\Rightarrow$ r(W,b), we infer r(a,b) under the substitution set {a/W, b/X}.

Another important concept is the *composition* of unification substitutions. If S and S′ are two substitution sets, then the composition of S and S′ (written SS′) is obtained by applying S′ to the elements of S and adding the result to S. Consider the example of composing the following three sets of substitutions:

{X/Y, W/Z}, {V/X}, {a/V, f(b)/W}.

Composing the third set, {a/V, f(b)/W}, with the second, {V/X}, produces:

{a/X, a/V, f(b)/W}.

Composing this result with the first set, {X/Y, W/Z}, produces the set of substitutions:

{a/Y, a/X, a/V, f(b)/Z, f(b)/W}.

Composition is the method by which unification substitutions are combined and returned in the recursive function unify, presented next. Composition is associative but not commutative. The exercises present these issues in more detail.

A further requirement of the unification algorithm is that the unifier be as general as possible: that the *most general unifier* be found. This is important, as will be seen in the next example, because, if generality is lost in the solution process, it may lessen the scope of the eventual solution or even eliminate the possibility of a solution entirely.

For example, in unifying p(X) and p(Y) any constant expression such as {fred/X, fred/Y} will work. However, fred is not the most general unifier; any variable would produce a more general expression: {Z/X, Z/Y}. The solutions obtained from the first substitution instance would always be restricted by having the constant fred limit the resulting inferences; i.e., fred would be a unifier, but it would lessen the generality of the result.

DEFINITION

MOST GENERAL UNIFIER (mgu)

If s is any unifier of expressions E, and g is the most general unifier of that set of expressions, then for s applied to E there exists another unifier s′ such that Es = Egs′, where Es and Egs′ are the composition of unifiers applied to the expression E.

The most general unifier for a set of expressions is unique except for alphabetic variations; i.e., whether a variable is eventually called X or Y really does not make any difference to the generality of the resulting unifications.

Unification is important for any artificial intelligence problem solver that uses the predicate calculus for representation. Unification specifies conditions under which two (or more) predicate calculus expressions may be said to be equivalent. This allows use of inference rules, such as *resolution*, with logic representations, a process that often requires backtracking to find all possible interpretations.

We next present pseudo-code for a function, unify, that can compute the unifying substitutions (when this is possible) between two predicate calculus expressions. Unify takes as arguments two expressions in the predicate calculus and returns either the most general set of unifying substitutions or the constant FAIL if no unification is possible. It is defined as a recursive function: first, it recursively attempts to unify the initial components of the expressions. If this succeeds, any substitutions returned by this unification are applied to the remainder of both expressions. These are then passed in a second recursive call to

unify, which attempts to complete the unification. The recursion stops when either argument is a symbol (a predicate, function name, constant, or variable) or the elements of the expression have all been matched.

To simplify the manipulation of expressions, the algorithm assumes a slightly modified syntax. Because unify simply performs syntactic pattern matching, it can effectively ignore the predicate calculus distinction between predicates, functions, and arguments. By representing an expression as a *list* (an ordered sequence of elements) with the predicate or function name as the first element followed by its arguments, we simplify the manipulation of expressions. Expressions in which an argument is itself a predicate or function expression are represented as lists within the list, thus preserving the structure of the expression. Lists are delimited by parentheses, ( ), and list elements are separated by spaces. Examples of expressions in both predicate calculus, PC, and list syntax include:

| PC SYNTAX | LIST SYNTAX |
|---|---|
| p(a,b) | (p a b) |
| p(f(a),g(X,Y)) | (p (f a) (g X Y)) |
| equal(eve,mother(cain)) | (equal eve (mother cain)) |

We next present the function unify:

```
function unify(E1, E2);
  begin
    case
      both E1 and E2 are constants or the empty list:                %recursion stops
        if E1 = E2 then return {}
          else return FAIL;
      E1 is a variable:
        if E1 occurs in E2 then return FAIL
          else return {E2/E1};
      E2 is a variable:
        if E2 occurs in E1 then return FAIL
          else return {E1/E2}
      either E1 or E2 are empty then return FAIL     %the lists are of different sizes
      otherwise:                                            %both E1 and E2 are lists
        begin
            HE1 := first element of E1;
            HE2 := first element of E2;
            SUBS1 := unify(HE1,HE2);
            if SUBS1 : = FAIL then return FAIL;
            TE1 := apply(SUBS1, rest of E1);
            TE2 : = apply (SUBS1, rest of E2);
            SUBS2 : = unify(TE1, TE2);
            if SUBS2 = FAIL then return FAIL;
                else return composition(SUBS1,SUBS2)
        end
    end                                                                   %end case
  end
```

### 2.3.3 A Unification Example

The behavior of the preceding algorithm may be clarified by tracing the call

unify((parents X (father X) (mother bill)), (parents bill (father bill) Y)).

When unify is first called, because neither argument is an atomic symbol, the function will attempt to recursively unify the first elements of each expression, calling

unify(parents, parents).

This unification succeeds, returning the empty substitution, { }. Applying this to the remainder of the expressions creates no change; the algorithm then calls

unify((X (father X) (mother bill)), (bill (father bill) Y)).

A tree depiction of the execution at this stage appears in Figure 2.4.

In the second call to unify, neither expression is atomic, so the algorithm separates each expression into its first component and the remainder of the expression. This leads to the call

unify(X, bill).

This call succeeds, because both expressions are atomic and one of them is a variable. The call returns the substitution {bill/X}. This substitution is applied to the remainder of each expression and unify is called on the results, as in Figure 2.5:

unify(((father bill) (mother bill)), ((father bill)Y)).

The result of this call is to unify (father bill) with (father bill). This leads to the calls

unify(father, father)
unify(bill, bill)
unify(( ), ( ))

All of these succeed, returning the empty set of substitutions as seen in Figure 2.6.

Unify is then called on the remainder of the expressions:

unify(((mother bill)), (Y)).

This, in turn, leads to calls

unify((mother bill), Y)

unify(( ),( )).

In the first of these, (mother bill) unifies with Y. Notice that unification substitutes the whole *structure* (mother bill) for the variable Y. Thus, unification succeeds and returns the substitution {(mother bill)/Y}. The call

```
unify(( ),( ))
```

returns { }. All the substitutions are composed as each recursive call terminates, to return the answer {bill/X (mother bill)/Y}. A trace of the entire execution appears in Figure 2.6. Each call is numbered to indicate the order in which it was made; the substitutions returned by each call are noted on the arcs of the tree.

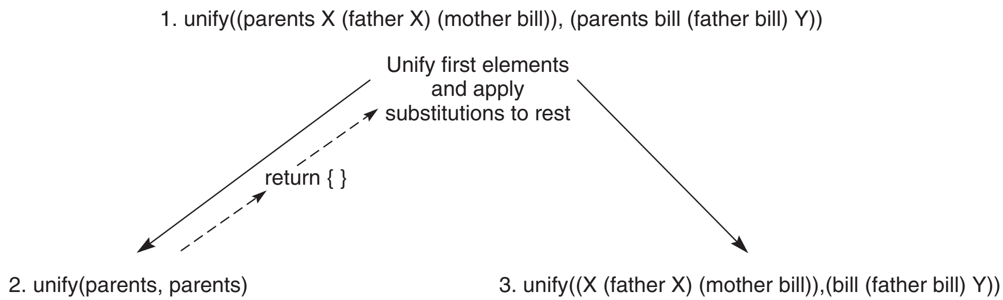



Figure 2.4 Initial steps in the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).

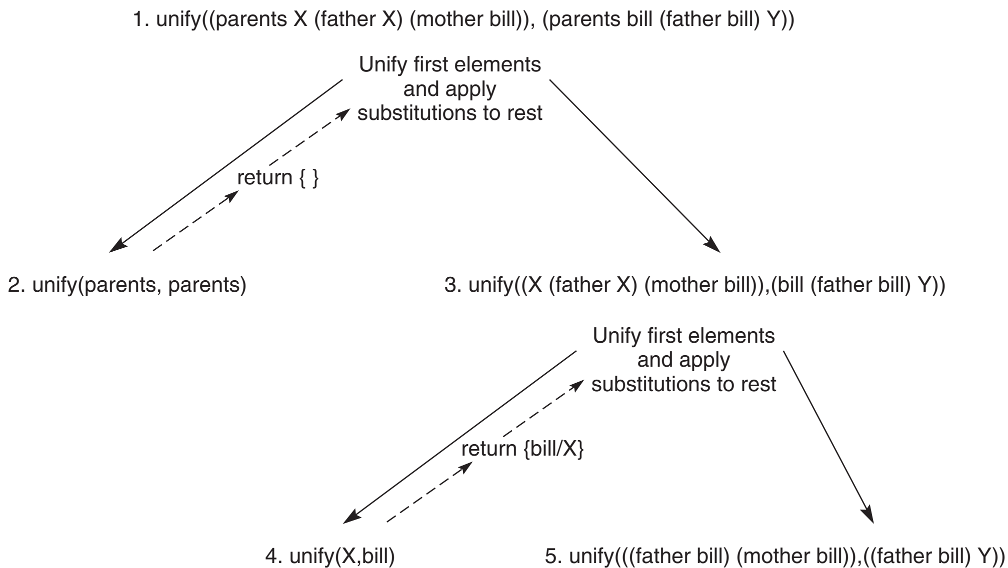



Figure 2.5 Further steps in the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).

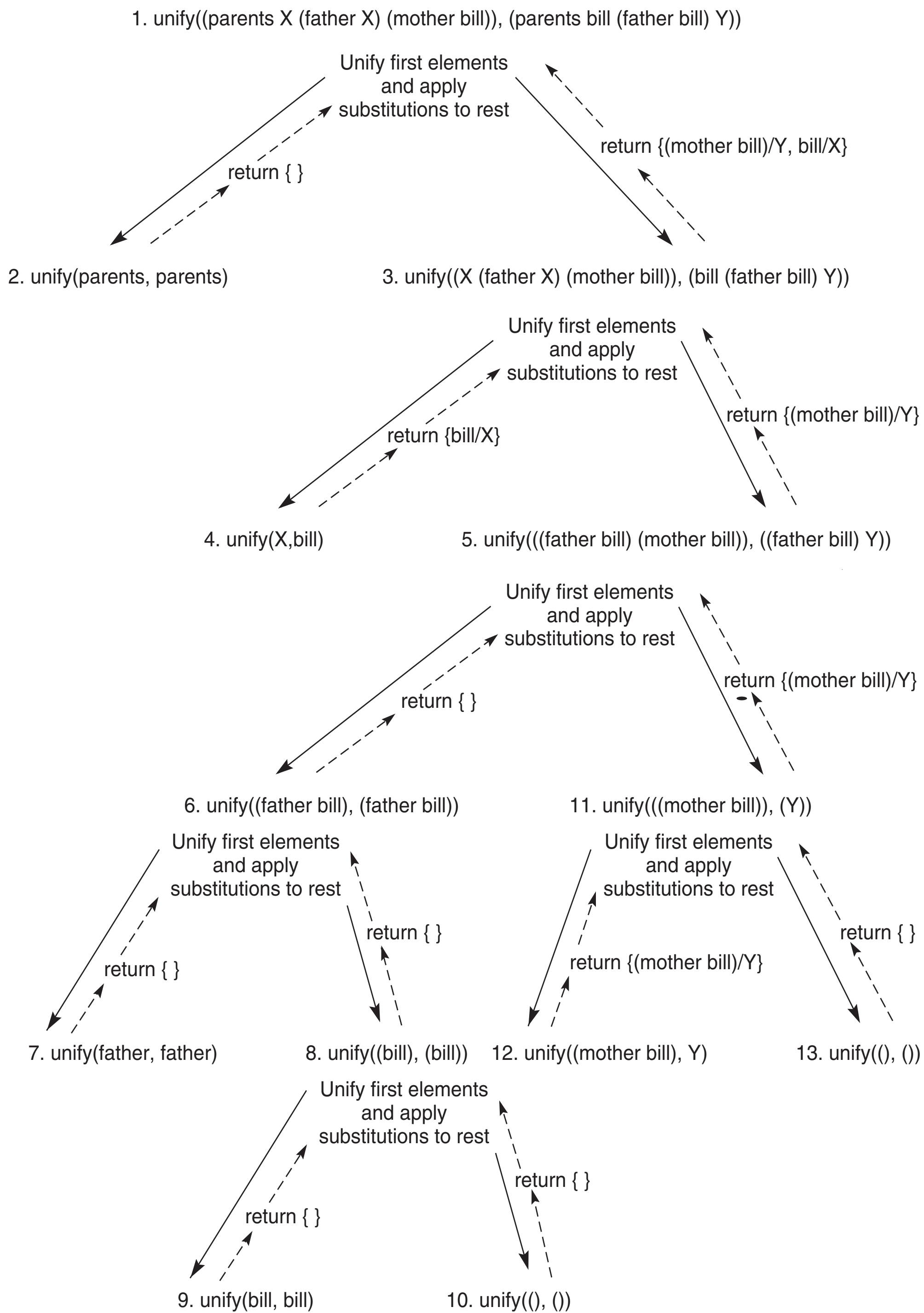



Figure 2.6 Final trace of the unification of (parents X (father X) (mother bill)) and (parents bill (father bill) Y).

| | — | e | x | e | c | u | t | i | o | n |
|---|---|---|---|---|---|---|---|---|---|---|
| — | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i | 1 | **2** | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| n | 2 | 3 | **4** | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| t | 3 | 4 | **5** | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| e | 4 | 5 | 6 | **5** | **6** | 7 | 8 | 9 | 10 | 11 |
| n | 5 | 6 | 7 | 6 | 7 | **8** | 9 | 10 | 11 | 12 |
| t | 6 | 7 | 8 | 7 | 8 | 9 | **8** | 9 | 10 | 11 |
| i | 7 | 8 | 9 | 8 | 9 | 10 | 9 | **8** | 9 | 10 |
| o | 8 | 9 | 10 | 9 | 10 | 11 | 10 | 9 | **8** | 9 |
| n | 9 | 10 | 11 | 10 | 11 | 12 | 11 | 10 | 9 | **8** |

Figure 4.9 Complete array of minimum edit difference between intention and execution (adapted from Jurafsky and Martin 2000).

In the spell check situation of proposing a cost-based ordered list of words for replacing an unrecognized string, the backward segment of the dynamic programming algorithm is not needed. Once the minimum edit measure is calculated for the set of related strings a prioritized order of alternatives is proposed from which the user chooses an appropriate string.

The justification for dynamic programming is the cost of time/space in computation. Dynamic programming, as seen in our examples has cost of $n^2$, where $n$ is the length of the largest string; the cost in the worse case is $n^3$, if other related subproblems need to be considered (other row/column values) to determine the current state. Exhaustive search for comparing two strings is exponential, costing between $2^n$ and $3^n$.

There are a number of obvious heuristics that can be used to prune the search in dynamic programming. First, useful solutions will usually lie around the upper left to lower right diagonal of the array; this leads to ignoring development of array extremes. Second, it can be useful to prune the search as it evolves, e.g., for edit distances passing a certain threshold, cut that solution path or even abandon the whole problem, i.e., the source string will be so distant from the target string of characters as to be useless. There is also a stochastic approach to the pattern comparison problem that we will see in Section 5.3.

## 4.2 The Best-First Search Algorithm

### 4.2.1 Implementing Best-First Search

In spite of their limitations, algorithms such as backtrack, hill climbing, and dynamic programming can be used effectively if their evaluation functions are sufficiently informative to avoid local maxima, dead ends, and related anomalies in a search space. In general,

however, use of heuristic search requires a more flexible algorithm: this is provided by *best-first search*, where, with a priority queue, recovery from these situations is possible.

Like the depth-first and breadth-first search algorithms of Chapter 3, best-first search uses lists to maintain states: open to keep track of the current fringe of the search and closed to record states already visited. An added step in the algorithm orders the states on open according to some heuristic estimate of their "closeness" to a goal. Thus, each iteration of the loop considers the most "promising" state on the open list. The pseudo-code for the function best_first_search appears below.

```
function best_first_search;

begin
  open := [Start];                                              % initialize
  closed := [ ];
  while open ≠ [ ] do                                           % states remain
    begin
      remove the leftmost state from open, call it X;
      if X = goal then return the path from Start to X
      else begin
            generate children of X;
            for each child of X do
            case
                the child is not on open or closed:
                  begin
                      assign the child a heuristic value;
                      add the child to open
                  end;
                the child is already on open:
                  if the child was reached by a shorter path
                  then give the state on open the shorter path
                the child is already on closed:
                  if the child was reached by a shorter path then
                    begin
                      remove the state from closed;
                      add the child to open
                    end;
            end;                                                % case
            put X on closed;
            re-order states on open by heuristic merit (best leftmost)
          end;
  return FAIL                                                   % open is empty
end.
```

At each iteration, best_first_search removes the first element from the open list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Note that each state retains ancestor information to determine if it had previously been reached by a shorter path and to allow the algorithm to return the final solution path. (See Section 3.2.3.)

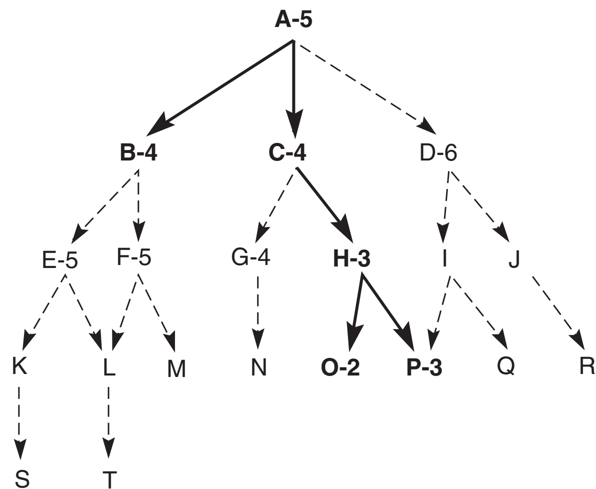



Figure 4.10 Heuristic search of a hypothetical state space.

If the first element on open is not a goal, the algorithm applies all matching production rules or operators to generate its descendants. If a child state is not on open or closed best_first_search applies a heuristic evaluation to that state, and the open list is sorted according to the heuristic values of those states. This brings the "best" states to the front of open. Note that because these estimates are heuristic in nature, the next "best" state to be examined may be from any level of the state space. When open is maintained as a sorted list, it is often referred to as a *priority queue*.

If a child state is already on open or closed, the algorithm checks to make sure that the state records the shorter of the two partial solution paths. Duplicate states are not retained. By updating the path history of nodes on open and closed when they are rediscovered, the algorithm will find a shortest path to a goal (within the states considered).

Figure 4.10 shows a hypothetical state space with heuristic evaluations attached to some of its states. The states with attached evaluations are those actually generated in best_first_search. The states expanded by the heuristic search algorithm are indicated in bold; note that it does not search all of the space. The goal of best-first search is to find the goal state by looking at as few states as possible; the more *informed* (Section 4.2.3) the heuristic, the fewer states are processed in finding the goal.

A trace of the execution of best_first_search on this graph appears below. Suppose P is the goal state in the graph of Figure 4.10. Because P is the goal, states along the path to P will tend to have lower heuristic values. The heuristic is fallible: the state O has a lower value than the goal itself and is examined first. Unlike hill climbing, which does not maintain a priority queue for the selection of "next" states, the algorithm recovers from this error and finds the correct goal.

1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]

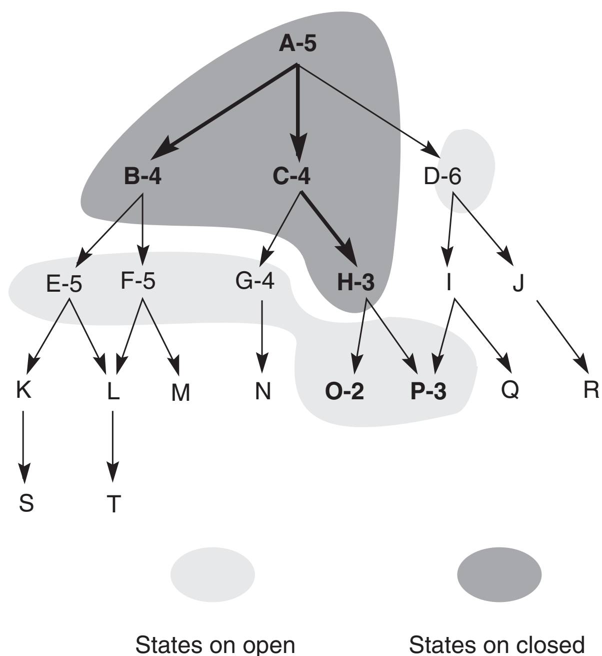



Figure 4.11 Heuristic search of a hypothetical state space with open and closed states highlighted.

3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!

Figure 4.11 shows the space as it appears after the fifth iteration of the while loop. The states contained in open and closed are indicated. open records the current frontier of the search and closed records states already considered. Note that the frontier of the search is highly uneven, reflecting the opportunistic nature of best-first search.

The best-first search algorithm selects the most promising state on open for further expansion. However, as it is using a heuristic that may prove erroneous, it does not abandon all the other states but maintains them on open. In the event a heuristic leads the search down a path that proves incorrect, the algorithm will eventually retrieve some previously generated, “next best” state from open and shift its focus to another part of the space. In the example of Figure 4.10, after the children of state B were found to have poor heuristic evaluations, the search shifted its focus to state C. The children of B were kept on open in case the algorithm needed to return to them later. In best_first_search, as in the algorithms of Chapter 3, the open list supports backtracking from paths that fail to produce a goal.

### 4.2.2 Implementing Heuristic Evaluation Functions

We next evaluate the performance of several different heuristics for solving the 8-puzzle. Figure 4.12 shows a start and goal state for the 8-puzzle, along with the first three states generated in the search.

The simplest heuristic counts the tiles out of place in each state when compared with the goal. This is intuitively appealing, because it would seem that, all else being equal, the state that had fewest tiles out of place is probably closer to the desired goal and would be the best to examine next.

However, this heuristic does not use all of the information available in a board configuration, because it does not take into account the distance the tiles must be moved. A "better" heuristic would sum all the distances by which the tiles are out of place, one for each square a tile must be moved to reach its position in the goal state.

Both of these heuristics can be criticized for failing to acknowledge the difficulty of tile reversals. That is, if two tiles are next to each other and the goal requires their being in opposite locations, it takes (several) more than two moves to put them back in place, as the tiles must "go around" each other (Figure 4.13).

A heuristic that takes this into account multiplies a small number (2, for example) times each direct tile reversal (where two adjacent tiles must be exchanged to be in the order of the goal). Figure 4.14 shows the result of applying each of these three heuristics to the three child states of Figure 4.12.

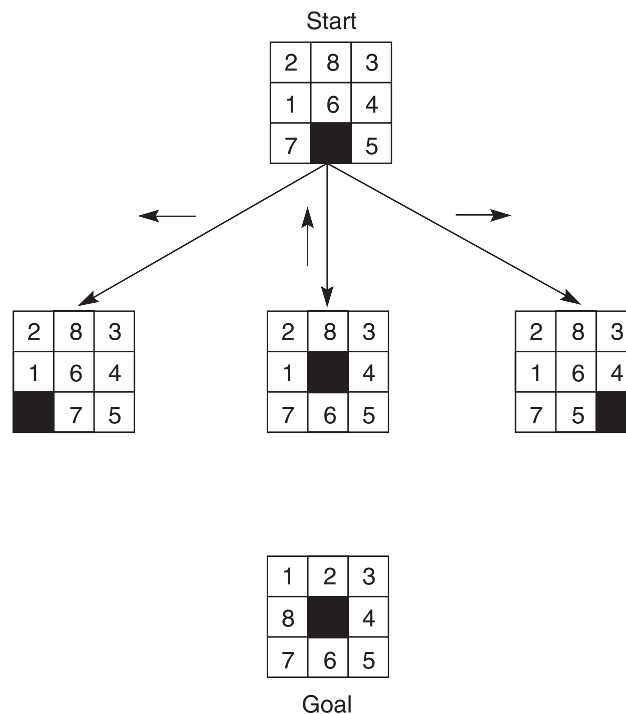



Figure 4.12 The start state, first moves, and goal state for an example 8-puzzle.

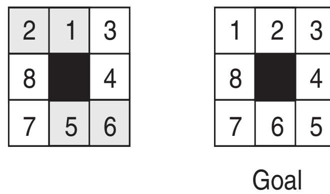



Figure 4.13 An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.

In Figure 4.14's summary of evaluation functions, the sum of distances heuristic does indeed seem to provide a more accurate estimate of the work to be done than the simple count of the number of tiles out of place. Also, the tile reversal heuristic fails to distinguish between these states, giving each an evaluation of 0. Although it is an intuitively appealing heuristic, it breaks down since none of these states have any direct reversals. A fourth heuristic, which may overcome the limitations of the tile reversal heuristic, adds the sum of the distances the tiles are out of place and 2 times the number of direct reversals.

This example illustrates the difficulty of devising good heuristics. Our goal is to use the limited information available in a single state descriptor to make intelligent choices. Each of the heuristics proposed above ignores some critical bit of information and is subject to improvement. The design of good heuristics is an empirical problem; judgment and intuition help, but the final measure of a heuristic must be its actual performance on problem instances.

If two states have the same or nearly the same heuristic evaluations, it is generally preferable to examine the state that is nearest to the root state of the graph. This state will have a greater probability of being on the *shortest* path to the goal. The distance from the

| | Tiles out of place | Sum of distances out of place | 2 x the number of direct tile reversals |
|---|---|---|---|
| 2 8 3<br>1 6 4<br>7 5 | **5** | **6** | **0** |
| 2 8 3<br>1 4<br>7 6 5 | **3** | **4** | **0** |
| 2 8 3<br>1 6 4<br>7 5 | **5** | **6** | **0** |

Goal:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

Figure 4.14 Three heuristics applied to states in the 8-puzzle.

starting state to its descendants can be measured by maintaining a depth count for each state. This count is 0 for the beginning state and is incremented by 1 for each level of the search. This depth measure can be added to the heuristic evaluation of each state to bias search in favor of states found shallower in the graph.

This makes our evaluation function, f, the sum of two components:

$$f(n) = g(n) + h(n)$$

where g(n) measures the actual length of the path from any state n to the start state and h(n) is a heuristic estimate of the distance from state n to a goal.

In the 8-puzzle, for example, we can let h(n) be the number of tiles out of place. When this evaluation is applied to each of the child states in Figure 4.12, their f values are 6, 4, and 6, respectively, see Figure 4.15.

The full best-first search of the 8-puzzle graph, using f as defined above, appears in Figure 4.16. Each state is labeled with a letter and its heuristic weight, f(n) = g(n) + h(n). The number at the top of each state indicates the order in which it was taken off the open list. Some states (h, g, b, d, n, k, and i) are not so numbered, because they were still on open when the algorithm terminates.

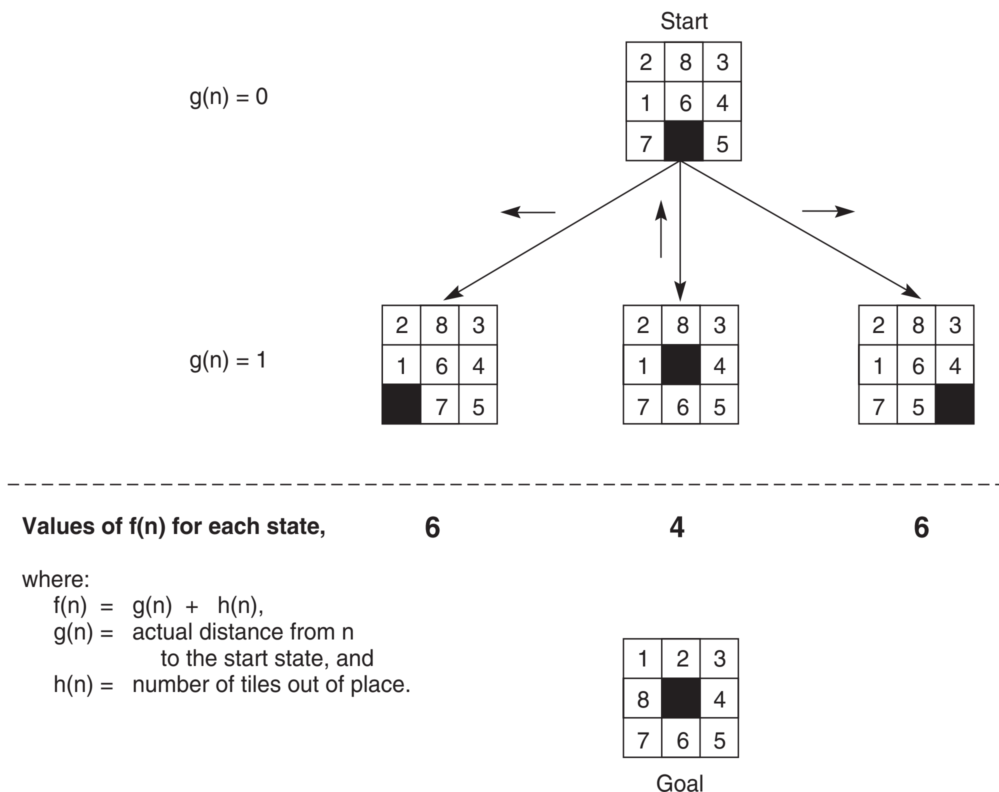



Figure 4.15 The heuristic f applied to states in the 8-puzzle.

The successive stages of open and closed that generate this graph are:

1. open = [a4];
   closed = [ ]
2. open = [c4, b6, d6];
   closed = [a4]
3. open = [e5, f5, b6, d6, g6];
   closed = [a4, c4]
4. open = [f5, h6, b6, d6, g6, i7];
   closed = [a4, c4, e5]
5. open = [ j5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5]
6. open = [l5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5, j5]
7. open = [m5, h6, b6, d6, g6, n7, k7, i7];
   closed = [a4, c4, e5, f5, j5, l5]
8. success, m = goal!

In step 3, both e and f have a heuristic of 5. State e is examined first, producing children, h and i. Although h, the child of e, has the same number of tiles out of place as f, it is one level deeper in the space. The depth measure, g(n), causes the algorithm to select f for evaluation in step 4. The algorithm goes back to the shallower state and continues to the goal. The state space graph at this stage of the search, with open and closed highlighted, appears in Figure 4.17. Notice the opportunistic nature of best-first search.

In effect, the g(n) component of the evaluation function gives the search more of a breadth-first flavor. This prevents it from being misled by an erroneous evaluation: if a heuristic continuously returns "good" evaluations for states along a path that fails to reach a goal, the g value will grow to dominate h and force search back to a shorter solution path. This guarantees that the algorithm will not become permanently lost, descending an infinite branch. Section 4.3 examines the conditions under which best-first search using this evaluation function can actually be guaranteed to produce the shortest path to a goal.

To summarize:

1. Operations on states generate children of the state currently under examination.
2. Each new state is checked to see whether it has occurred before (is on either open or closed), thereby preventing loops.
3. Each state n is given an f value equal to the sum of its depth in the search space g(n) and a heuristic estimate of its distance to a goal h(n). The h value guides search toward heuristically promising states while the g value can prevent search from persisting indefinitely on a fruitless path.

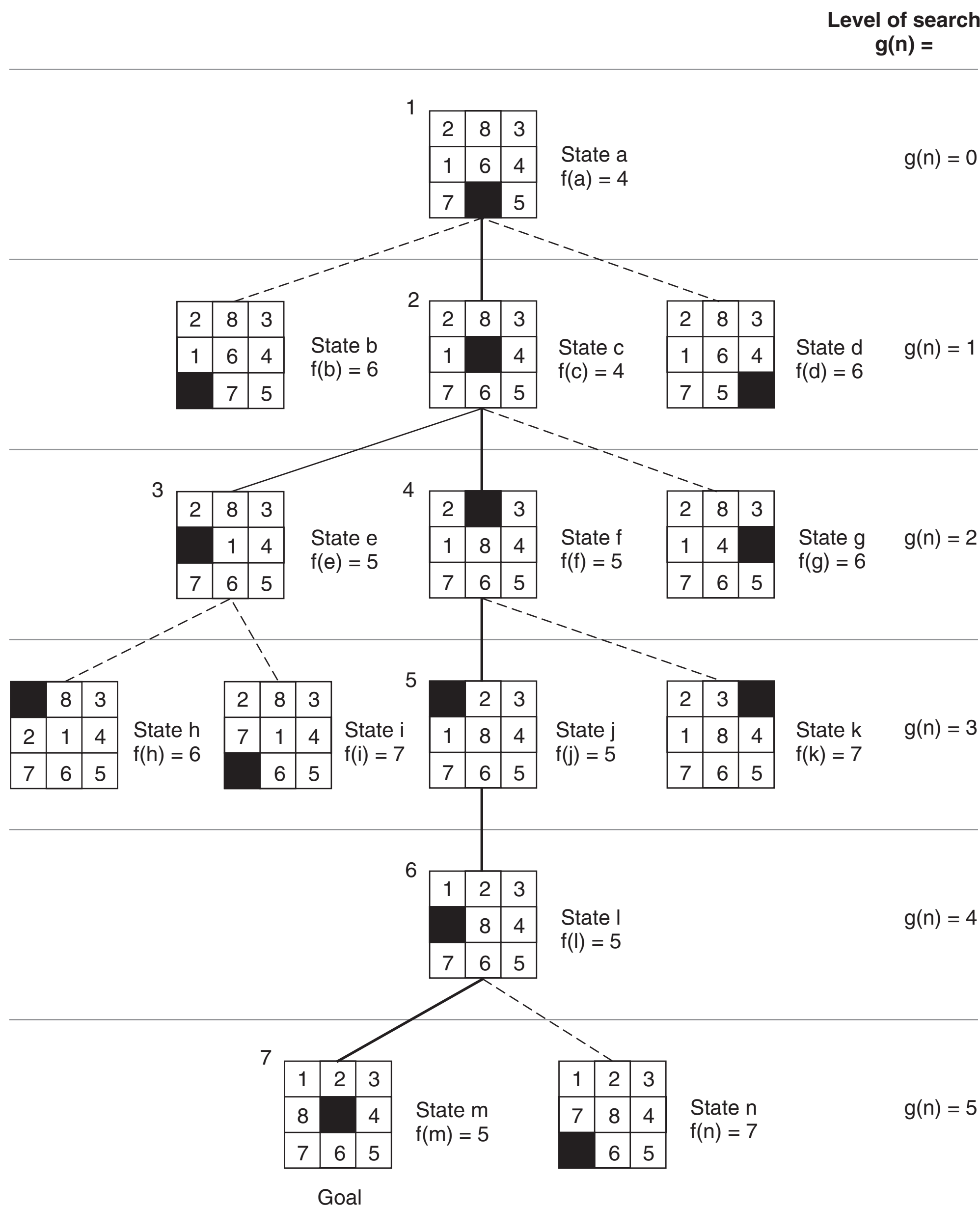



Figure 4.16 State space generated in heuristic search of the 8-puzzle graph.

4. States on open are sorted by their f values. By keeping all states on open until they are examined or a goal is found, the algorithm recovers from dead ends.

5. As an implementation point, the algorithm's efficiency can be improved through maintenance of the open and closed lists, perhaps as *heaps* or *leftist trees.*

Best-first search is a general algorithm for heuristically searching any state space graph (as were the breadth- and depth-first algorithms presented earlier). It is equally

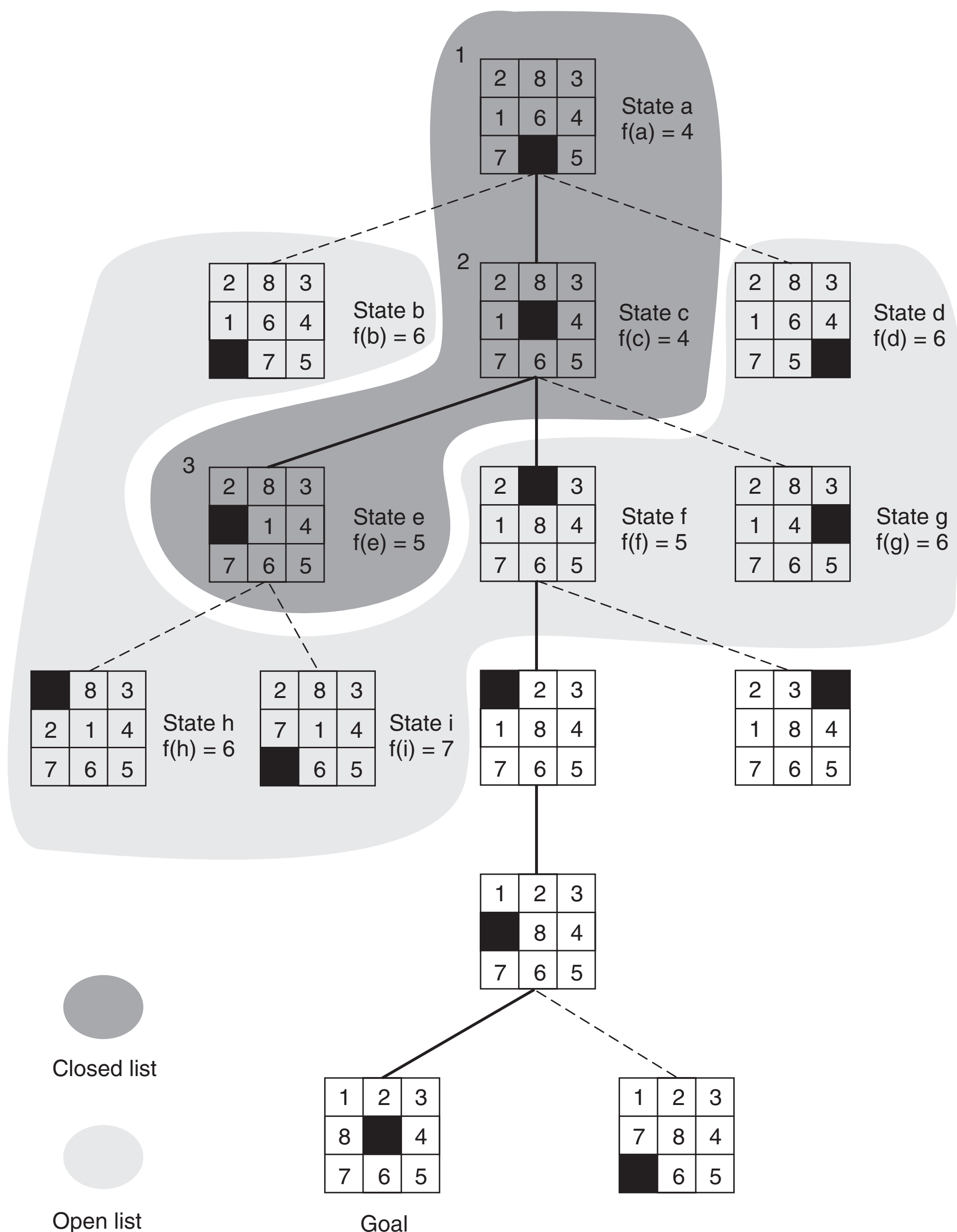



Figure 4.17 open and closed as they appear after the third iteration of heuristic search.

applicable to data- and goal-driven searches and supports a variety of heuristic evaluation functions. It will continue (Section 4.3) to provide a basis for examining the behavior of heuristic search. Because of its generality, best-first search can be used with a variety of heuristics, ranging from subjective estimates of state's "goodness" to sophisticated

measures based on the probability of a state leading to a goal. Bayesian statistical measures (Chapters 5 and 9) offer an important example of this approach.

Another interesting approach to implementing heuristics is the use of confidence measures by expert systems to weigh the results of a rule. When human experts employ a heuristic, they are usually able to give some estimate of their confidence in its conclusions. Expert systems employ *confidence measures* to select the conclusions with the highest likelihood of success. States with extremely low confidences can be eliminated entirely. This approach to heuristic search is examined in the next section.

### 4.2.3 Heuristic Search and Expert Systems

Simple games such as the 8-puzzle are ideal vehicles for exploring the design and behavior of heuristic search algorithms for a number of reasons:

1. The search spaces are large enough to require heuristic pruning.
2. Most games are complex enough to suggest a rich variety of heuristic evaluations for comparison and analysis.
3. Games generally do not involve complex representational issues. A single node of the state space is just a board description and usually can be captured in a straightforward fashion. This allows researchers to focus on the behavior of the heuristic rather than the problems of knowledge representation.
4. Because each node of the state space has a common representation (e.g., a board description), a single heuristic may be applied throughout the search space. This contrasts with systems such as the financial advisor, where each node represents a different subgoal with its own distinct description.

More realistic problems greatly complicate the implementation and analysis of heuristic search by requiring multiple heuristics to deal with different situations in the problem space. However, the insights gained from simple games generalize to problems such as those found in expert systems applications, planning, intelligent control, and machine learning. Unlike the 8-puzzle, a single heuristic may not apply to each state in these domains. Instead, situation specific problem-solving heuristics are encoded in the syntax and content of individual problem solving operators. Each solution step incorporates its own heuristic that determines when it should be applied; the pattern matcher matches the appropriate operation (heuristic) with the relevant state in the space.

**EXAMPLE 4.2.1: THE FINANCIAL ADVISOR, REVISITED**

The use of heuristic measures to guide search is a general approach in AI. Consider again the financial advisor problem of Chapters 2 and 3, where the knowledge base was treated as a set of logical implications, whose conclusions are either true or false. In actuality, these rules are highly heuristic in nature. For example, one rule states that an individual with adequate savings and income should invest in stocks:

# 3 STRUCTURES AND STRATEGIES FOR STATE SPACE SEARCH

*In order to cope, an organism must either armor itself (like a tree or a clam) and "hope for the best," or else develop methods for getting out of harm's way and into the better neighborhoods of the vicinity. If you follow this later course, you are confronted with the primordial problem that every agent must continually solve: Now what do I do?*

—Daniel C. Dennett, *"Consciousness Explained"*

*Two roads diverged in a yellow wood,*
*And sorry I could not travel both*
*And be one traveler, long I stood*
*And looked down one as far as I could*
*To where it bent in the undergrowth;*
*Then took the other. . .*

—Robert Frost, *"The Road Not Taken"*

## 3.0 Introduction

Chapter 2 introduced predicate calculus as an example of an artificial intelligence representation language. Well-formed predicate calculus expressions provide a means of describing objects and relations in a problem domain, and inference rules such as modus ponens allow us to infer new knowledge from these descriptions. These inferences define a space that is searched to find a problem solution. Chapter 3 introduces the theory of state space search.

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? 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?

The theory of *state space search* is our primary tool for answering these questions. By representing a problem as a *state space graph*, we can use *graph theory* to analyze the structure and complexity of both the problem and the search procedures that we employ to solve it.

A graph consists of a set of *nodes* and a set of *arcs* or *links* connecting pairs of nodes. In the state space model of problem solving, the nodes of a graph are taken to represent discrete *states* in a problem-solving process, such as the results of logical inferences or the different configurations of a game board. The arcs of the graph represent transitions between states. These transitions correspond to logical inferences or legal moves of a game. In expert systems, for example, states describe our knowledge of a problem instance at some stage of a reasoning process. Expert knowledge, in the form of *if . . . then* rules, allows us to generate new information; the act of applying a rule is represented as an arc between states.

Graph theory is our best tool for reasoning about the structure of objects and relations; indeed, this is precisely the need that led to its creation in the early eighteenth century. The Swiss mathematician Leonhard Euler invented graph theory to solve the "bridges of Königsberg problem." The city of Königsberg occupied both banks and two islands of a river. The islands and the riverbanks were connected by seven bridges, as indicated in Figure 3.1.

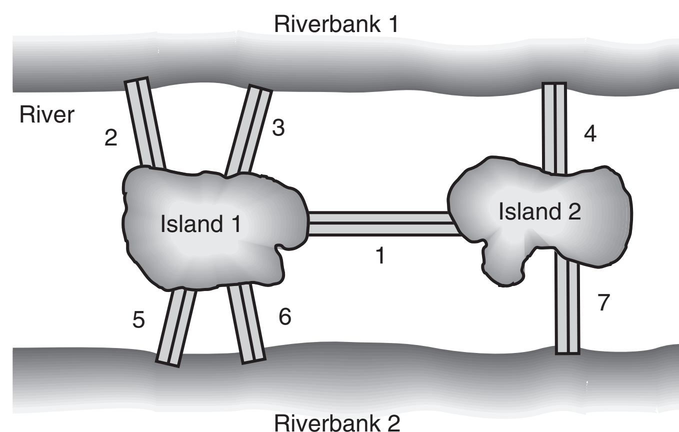



Figure 3.1 The city of Königsberg.

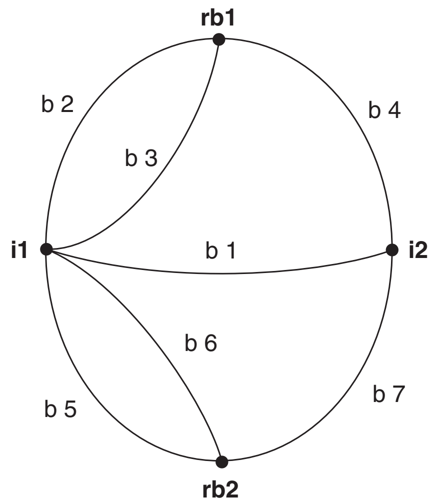



Figure 3.2 Graph of the Königsberg bridge system.

The bridges of Königsberg problem asks if there is a walk around the city that crosses each bridge exactly once. Although the residents had failed to find such a walk and doubted that it was possible, no one had proved its impossibility. Devising a form of graph theory, Euler created an alternative representation for the map, presented in Figure 3.2. The riverbanks (rb1 and rb2) and islands (i1 and i2) are described by the nodes of a graph; the bridges are represented by labeled arcs between nodes (b1, b2, , b7 ). The graph representation preserves the essential structure of the bridge system, while ignoring extraneous features such as bridge lengths, distances, and order of bridges in the walk.

Alternatively, we may represent the Königsberg bridge system using predicate calculus. The connect predicate corresponds to an arc of the graph, asserting that two land masses are connected by a particular bridge. Each bridge requires two connect predicates, one for each direction in which the bridge may be crossed. A predicate expression, connect(X, Y, Z) = connect (Y, X, Z), indicating that any bridge can be crossed in either direction, would allow removal of half the following connect facts:

connect(i1, i2, b1) connect(i2, i1, b1)

connect(rb1, i1, b2) connect(i1, rb1, b2)

connect(rb1, i1, b3) connect(i1, rb1, b3)

connect(rb1, i2, b4) connect(i2, rb1, b4)

connect(rb2, i1, b5) connect(i1, rb2, b5)

connect(rb2, i1, b6) connect(i1, rb2, b6)

connect(rb2, i2, b7) connect(i2, rb2, b7)

The predicate calculus representation is equivalent to the graph representation in that the connectedness is preserved. Indeed, an algorithm could translate between the two representations with no loss of information. However, the structure of the problem can be visualized more directly in the graph representation, whereas it is left implicit in the predicate calculus version. Euler's proof illustrates this distinction.

In proving that the walk was impossible, Euler focused on the *degree* of the nodes of the graph, observing that a node could be of either *even* or *odd* degree. An *even* degree node has an even number of arcs joining it to neighboring nodes. An *odd* degree node has an odd number of arcs. With the exception of its beginning and ending nodes, the desired walk would have to leave each node exactly as often as it entered it. Nodes of odd degree could be used only as the beginning or ending of the walk, because such nodes could be crossed only a certain number of times before they proved to be a dead end. The traveler could not exit the node without using a previously traveled arc.

Euler noted that unless a graph contained either exactly zero or two nodes of odd degree, the walk was impossible. If there were two odd-degree nodes, the walk could start at the first and end at the second; if there were no nodes of odd degree, the walk could begin and end at the same node. The walk is not possible for graphs containing any other number of nodes of odd degree, as is the case with the city of Königsberg. This problem is now called finding an *Euler path* through a graph.

Note that the predicate calculus representation, though it captures the relationships between bridges and land in the city, does not suggest the concept of the degree of a node. In the graph representation there is a single instance of each node with arcs between the nodes, rather than multiple occurrences of constants as arguments in a set of predicates. For this reason, the graph representation suggests the concept of node degree and the focus of Euler's proof. This illustrates graph theory's power for analyzing the structure of objects, properties, and relationships.

In Section 3.1 we review basic graph theory and then present finite state machines and the state space description of problems. In section 3.2 we introduce graph search as a problem-solving methodology. Depth-first and breadth-first search are two strategies for searching a state space. We compare these and make the added distinction between goal-driven and data-driven search. Section 3.3 demonstrates how state space search may be used to characterize reasoning with logic. Throughout the chapter, we use graph theory to analyze the structure and complexity of a variety of problems.

## 3.1 Structures for State Space Search

### 3.1.1 Graph Theory (optional)

A *graph* is a set of *nodes* or *states* and a set of *arcs* that connect the nodes. A *labeled* graph has one or more descriptors (labels) attached to each node that distinguish that node from any other node in the graph. In a *state space graph*, these descriptors identify states in a problem-solving process. If there are no descriptive differences between two nodes, they are considered the same. The arc between two nodes is indicated by the labels of the connected nodes.

The arcs of a graph may also be labeled. Arc labels are used to indicate that an arc represents a named relationship (as in a semantic network) or to attach weights to arcs (as in the traveling salesperson problem). If there are different arcs between the same two nodes (as in Figure 3.2), these can also be distinguished through labeling.

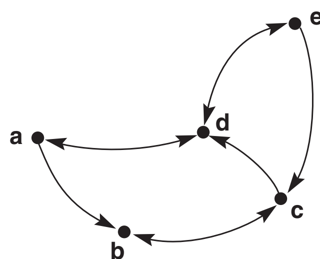
A graph is *directed* if arcs have an associated directionality. The arcs in a directed graph are usually drawn as arrows or have an arrow attached to indicate direction. Arcs that can be crossed in either direction may have two arrows attached but more often have no direction indicators at all. Figure 3.3 is a labeled, directed graph: arc (a, b) may only be crossed from node a to node b, but arc (b, c) is crossable in either direction.

A *path* through a graph connects a sequence of nodes through successive arcs. The path is represented by an ordered list that records the nodes in the order they occur in the path. In Figure 3.3, [a, b, c, d] represents the path through nodes a, b, c, and d, in that order.

A *rooted* graph has a unique node, called the *root*, such that there is a path from the root to all nodes within the graph. In drawing a rooted graph, the root is usually drawn at the top of the page, above the other nodes. The state space graphs for games are usually rooted graphs with the start of the game as the root. The initial moves of the tic-tac-toe game graph are represented by the rooted graph of Figure II.5. This is a directed graph with all arcs having a single direction. Note that this graph contains no cycles; players cannot (as much as they might sometimes wish!) undo a move.

A *tree* is a graph in which two nodes have at most one path between them. Trees often have roots, in which case they are usually drawn with the root at the top, like a rooted graph. Because each node in a tree has only one path of access from any other node, it is impossible for a path to *loop* or *cycle* through a sequence of nodes.

For rooted trees or graphs, relationships between nodes include *parent*, *child*, and *sibling*. These are used in the usual familial fashion with the parent preceding its child along a directed arc. The children of a node are called *siblings*. Similarly, an *ancestor* comes before a *descendant* in some path of a directed graph. In Figure 3.4, b is a *parent* of nodes e and f (which are, therefore, *children* of b and *siblings* of each other). Nodes a and c are *ancestors* of states g, h, and i, and g, h, and i are *descendants* of a and c.




Nodes = **{a,b,c,d,e}**

Arcs = **{(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}**

Figure 3.3 A labeled directed graph.

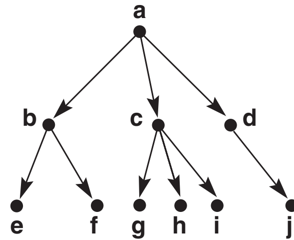



Figure 3.4 A rooted tree, exemplifying family relationships.

Before introducing the state space representation of problems we formally define these concepts.

DEFINITION

GRAPH

A graph consists of:

A set of *nodes* $N_1$, $N_2$, $N_3$, ..., $N_n$, ..., which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc ($N_3$, $N_4$) connects node $N_3$ to node $N_4$. This indicates a direct connection from node $N_3$ to $N_4$ but not from $N_4$ to $N_3$, unless ($N_4$, $N_3$) is also an arc, and then the arc joining $N_3$ and $N_4$ is *undirected*.

If a *directed* arc connects $N_j$ and $N_k$, then $N_j$ is called the *parent* of $N_k$ and $N_k$, the *child* of $N_j$. If the graph also contains an arc ($N_j$, $N_l$), then $N_k$ and $N_l$ are called *siblings*.

A *rooted* graph has a unique node $N_S$ from which all paths in the graph originate. That is, the root has no parent in the graph.

A *tip* or *leaf* node is a node that has no children.

An ordered sequence of nodes [$N_1$, $N_2$, $N_3$, ..., $N_n$], where each pair $N_i$, $N_{i+1}$ in the sequence represents an arc, i.e., ($N_i$, $N_{i+1}$), is called a *path* of length $n - 1$.

On a path in a rooted graph, a node is said to be an *ancestor* of all nodes positioned after it (to its right) as well as a *descendant* of all nodes before it.

A path that contains any node more than once (some $N_j$ in the definition of path above is repeated) is said to contain a *cycle* or *loop.*

A *tree* is a graph in which there is a unique path between every pair of nodes. (The paths in a tree, therefore, contain no cycles.)

The edges in a rooted tree are directed away from the root. Each node in a rooted tree has a unique parent.

Two nodes are said to be *connected* if a path exists that includes them both.

Next we introduce the finite state machine, an abstract representation for computational devices, that may be viewed as an automaton for traversing paths in a graph.

### 3.1.2 The Finite State Machine (optional)

We can think of a machine as a system that accepts input values, possibly produces output values, and that has some sort of internal mechanism (states) to keep track of information about previous input values. A *finite state machine* (FSM) is a finite, directed, connected graph, having a set of states, a set of input values, and a state transition function that describes the effect that the elements of the input stream have on the states of the graph. The stream of input values produces a path within the graph of the states of this finite machine. Thus the FSM can be seen as an abstract model of computation.

The primary use for such a machine is to recognize components of a formal language. These components are often strings of characters ("words" made from characters of an "alphabet"). In Section 5.3 we extend this definition to a probabilistic finite state machine. These state machines have an important role in analyzing expressions in languages, whether computational or human, as we see in Sections 5.3, 9.3, and Chapter 15.

DEFINITION

FINITE STATE MACHINE (FSM)

A *finite state machine* is an ordered triple $(S, I, F)$, where:

$S$ is a finite set of *states* in a connected graph $s_1, s_2, s_3, \ldots s_n$.

$I$ is a finite set of *input* values $i_1, i_2, i_3, \ldots i_m$.

$F$ is a state transition function that for any $i \in I$, describes its effect on the states $S$ of the machine, thus $\forall i \in I, F_i : (S \rightarrow S)$. If the machine is in state $s_j$ and input $i$ occurs, the next state of the machine will be $F_i(s_j)$.

For a simple example of a finite state machine, let $S = \{s_0, s_1\}$, $I = \{0,1\}$, $f_0(s_0) = s_0$, $f_0(s_1) = (s_1)$, $f_1(s_0) = s_1$, and $f_1(s_1) = s_0$. With this device, sometimes called a *flip-flop*, an input value of zero leaves the state unchanged, while input 1 changes the state of the machine. We may visualize this machine from two equivalent perspectives, as a finite graph with labelled, directed arcs, as in Figure 3.5a, or as a transition matrix, Figure 3.5b. In the transition matrix, input values are listed along the top row, the states are in the left-most column, and the output for an input applied to a state is at the intersection point.

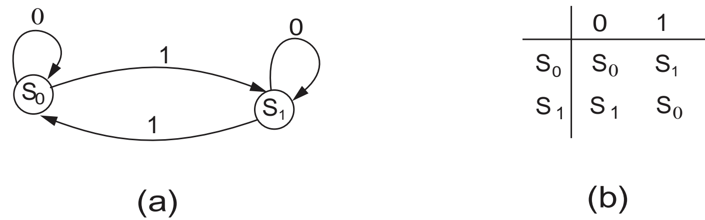



Figure 3.5 (a) The finite state graph for a flip-flop and (b) its transition matrix.

A second example of a finite state machine is represented by the directed graph of Figure 3.6a and the (equivalent) transition matrix of Figure 3.6b. One might ask what the finite state machine of Figure 3.6 could represent. With two assumptions, this machine could be seen as a recognizer of all strings of characters from the alphabet {a, b, c, d} that contain the exact sequence “abc”. The two assumptions are, first, that state $s_0$ has a special role as the *starting state*, and second, that $s_3$ is the *accepting state*. Thus, the input stream will present its first element to state $s_0$. If the stream later terminates with the machine in state $s_3$, it will have recognized that there is the sequence “abc” within that input stream.

What we have just described is a *finite state accepting machine*, sometimes called a *Moore machine*. We use the convention of placing an arrow from no state that terminates in the starting state of the Moore machine, and represent the accepting state (or states) as special, often using a doubled circle, as in Figure 3.6. We now present a formal definition of the Moore machine:

DEFINITION

FINITE STATE ACCEPTOR (MOORE MACHINE)

A *finite state acceptor* is a finite state machine (S, I, *F*), where:

$\exists s_0 \in S$ such that the input stream starts at $s_0$, and

$\exists s_n \in S$, an *accept* state. The input stream is accepted if it terminates in that state. In fact, there may be a set of accept states.

The finite state acceptor is represented as $(S, s_0, \{s_n\}, I, F)$

We have presented two fairly simple examples of a powerful concept. As we will see in natural language understanding (Chapter 15), finite state recognizers are an important tool for determining whether or not patterns of characters, words, or sentences have desired properties. We will see that a finite state accepter implicitly defines a formal language on the basis of the sets of letters (characters) and words (strings) that it accepts.

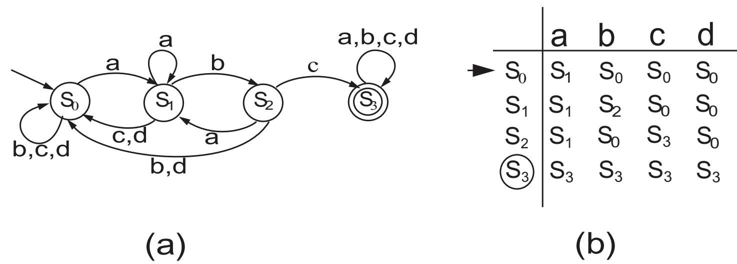



Figure 3.6 (a) The finite state graph and (b) the transition matrix for string recognition example.

We have also shown only *deterministic* finite state machines, where the transition function for any input value to a state gives a unique next state. *Probabilistic finite state machines*, where the transition function defines a distribution of output states for each input to a state, are also an important modeling technique. We consider these in Section 5.3 and again in Chapter 15. We next consider a more general graphical representation for the analysis of problem solving: the state space.

### 3.1.3 The State Space Representation of Problems

In the *state space representation* of a problem, the nodes of a graph correspond to partial problem solution *states* and the arcs correspond to steps in a problem-solving process. One or more *initial states*, corresponding to the given information in a problem instance, form the root of the graph. The graph also defines one or more *goal* conditions, which are solutions to a problem instance. *State space search* characterizes problem solving as the process of finding a *solution path* from the start state to a goal.

A goal may describe a state, such as a winning board in tic-tac-toe (Figure II.5) or a goal configuration in the 8-puzzle (Figure 3.7). Alternatively, a goal can describe some property of the solution path itself. In the traveling salesperson problem (Figures 3.9 and 3.10), search terminates when the "shortest" path is found through all nodes of the graph. In the parsing problem (Section 3.3), the solution path is a successful analysis of a sentence's structure.

Arcs of the state space correspond to steps in a solution process and paths through the space represent solutions in various stages of completion. Paths are searched, beginning at the start state and continuing through the graph, until either the goal description is satisfied or they are abandoned. The actual generation of new states along the path is done by applying operators, such as "legal moves" in a game or inference rules in a logic problem or expert system, to existing states on a path. The task of a search algorithm is to find a solution path through such a problem space. Search algorithms must keep track of the paths from a start to a goal node, because these paths contain the series of operations that lead to the problem solution.

We now formally define the state space representation of problems:

DEFINITION

STATE SPACE SEARCH

A *state space* is represented by a four-tuple [N,A,S,GD], where:

N is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.

A is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process.

S, a nonempty subset of N, contains the start state(s) of the problem.

GD, a nonempty subset of N, contains the goal state(s) of the problem. The states in GD are described using either:

1. A measurable property of the states encountered in the search.
2. A measurable property of the path developed in the search, for example, the sum of the transition costs for the arcs of the path.

A *solution path* is a path through this graph from a node in S to a node in GD.

One of the general features of a graph, and one of the problems that arise in the design of a graph search algorithm, is that states can sometimes be reached through different paths. For example, in Figure 3.3 a path can be made from state a to state d either through b and c or directly from a to d. This makes it important to choose the *best* path according to the needs of a problem. In addition, multiple paths to a state can lead to loops or cycles in a solution path that prevent the algorithm from reaching a goal. A blind search for goal state e in the graph of Figure 3.3 might search the sequence of states abcdabcdabcd . . . forever!

If the space to be searched is a tree, as in Figure 3.4, the problem of cycles does not occur. It is, therefore, important to distinguish between problems whose state space is a tree and those that may contain loops. General graph search algorithms must detect and eliminate loops from potential solution paths, whereas tree searches may gain efficiency by eliminating this test and its overhead.

Tic-tac-toe and the 8-puzzle exemplify the state spaces of simple games. Both of these examples demonstrate termination conditions of type 1 in our definition of state space search. Example 3.1.3, the traveling salesperson problem, has a goal description of type 2, the total cost of the path itself.

**EXAMPLE 3.1.1: TIC-TAC-TOE**

The state space representation of tic-tac-toe appears in Figure II.5, pg 42. The start state is an empty board, and the termination or goal description is a board state having three Xs in a row, column, or diagonal (assuming that the goal is a win for X). The path from the start state to a goal state gives the series of moves in a winning game.

The states in the space are all the different configurations of Xs and Os that the game can have. Of course, although there are $3^9$ ways to arrange {blank, X, O} in nine spaces, most of them would never occur in an actual game. Arcs are generated by legal moves of the game, alternating between placing an X and an O in an unused location. The state space is a graph rather than a tree, as some states on the third and deeper levels can be reached by different paths. However, there are no cycles in the state space, because the directed arcs of the graph do not allow a move to be undone. It is impossible to "go back up" the structure once a state has been reached. No checking for cycles in path generation is necessary. A graph structure with this property is called a *directed acyclic graph*, or *DAG*, and is common in state space search and in graphical models, Chapter 13.

The state space representation provides a means of determining the complexity of the problem. In tic-tac-toe, there are nine first moves with eight possible responses to each of them, followed by seven possible responses to each of these, and so on. It follows that $9 \times 8 \times 7 \times ...$ or 9! different paths can be generated. Although it is not impossible for a computer to search this number of paths (362,880) exhaustively, many important problems also exhibit factorial or exponential complexity, although on a much larger scale. Chess has $10^{120}$ possible game paths; checkers has $10^{40}$, some of which may never occur in an actual game. These spaces are difficult or impossible to search exhaustively. Strategies for searching such large spaces often rely on heuristics to reduce the complexity of the search (Chapter 4).

#### EXAMPLE 3.1.2: THE 8-PUZZLE

In the *15-puzzle* of Figure 3.7, 15 differently numbered tiles are fitted into 16 spaces on a grid. One space is left blank so that tiles can be moved around to form different patterns. The goal is to find a series of moves of tiles into the blank space that places the board in a goal configuration. This is a common game that most of us played as children. (The version I remember was about 3 inches square and had red and white tiles in a black frame.)

A number of interesting aspects of this game have made it useful to researchers in problem solving. The state space is large enough to be interesting but is not completely intractable (16! if symmetric states are treated as distinct). Game states are easy to represent. The game is rich enough to provide a number of interesting heuristics (see Chapter 4).

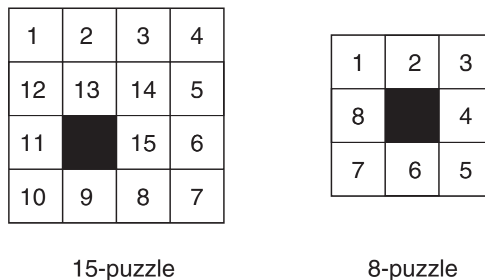



Figure 3.7 The 15-puzzle and the 8-puzzle.

The *8-puzzle* is a 3 × 3 version of the 15-puzzle in which eight tiles can be moved around in nine spaces. Because the 8-puzzle generates a smaller state space than the full 15-puzzle and its graph fits easily on a page, it is used for many examples in this book.

Although in the physical puzzle moves are made by moving tiles ("move the 7 tile right, provided the blank is to the right of the tile" or "move the 3 tile down"), it is much simpler to think in terms of "moving the blank space". This simplifies the definition of move rules because there are eight tiles but only a single blank. In order to apply a move, we must make sure that it does not move the blank off the board. Therefore, all four moves are not applicable at all times; for example, when the blank is in one of the corners only two moves are possible.

The legal moves are:

| | |
|---|---|
| move the blank up | ↑ |
| move the blank right | → |
| move the blank down | ↓ |
| move the blank left | ← |

If we specify a beginning state and a goal state for the 8-puzzle, it is possible to give a state space accounting of the problem-solving process (Figure 3.8). States could be

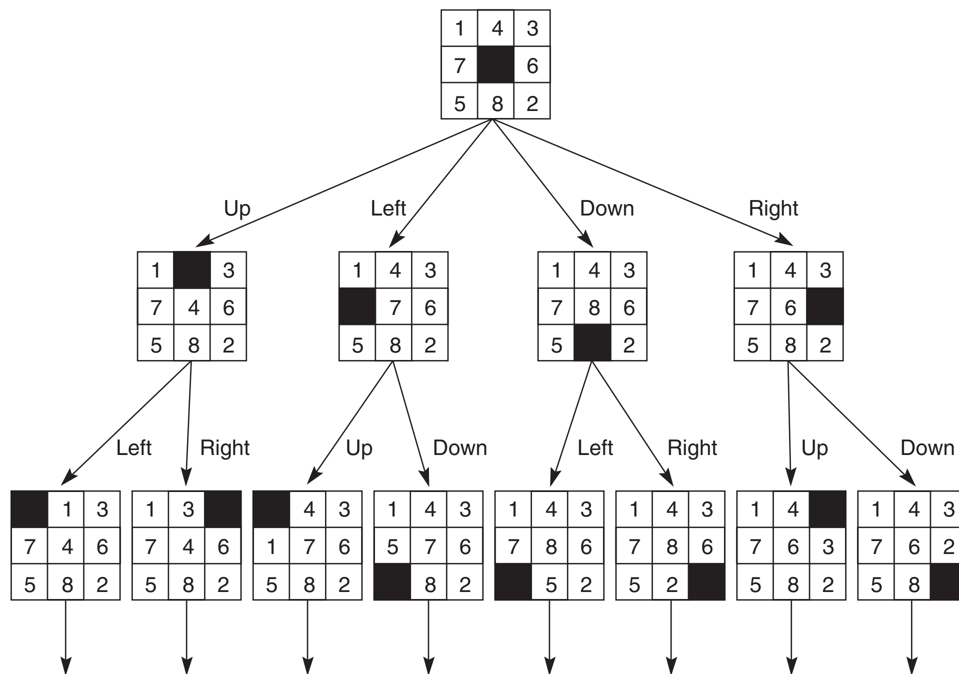



Figure 3.8 State space of the 8-puzzle generated by move blank operations.

represented using a simple 3 × 3 array. A predicate calculus representation could use a "state" predicate with nine parameters (for the locations of numbers in the grid). Four procedures, describing each of the possible moves of the blank, define the arcs in the state space.

As with tic-tac-toe, the state space for the 8-puzzle is a graph (with most states having multiple parents), but unlike tic-tac-toe, cycles are possible. The GD or goal description of the state space is a particular state or board configuration. When this state is found on a path, the search terminates. The path from start to goal is the desired series of moves.

It is interesting to note that the complete state space of the 8- and 15-puzzles consists of two disconnected (and in this case equal-sized) subgraphs. This makes half the possible states in the search space impossible to reach from any given start state. If we exchange (by prying loose!) two immediately adjacent tiles, states in the other component of the space become reachable.

#### EXAMPLE 3.1.3: THE TRAVELING SALESPERSON

Suppose a salesperson has five cities to visit and then must return home. The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city, and then returning to the starting city. Figure 3.9 gives an instance of this problem. The nodes of the graph represent cities, and each arc is labeled with a weight indicating the cost of traveling that arc. This cost might be a representation of the miles necessary in car travel or cost of an air flight between the two cities. For convenience, we assume the salesperson lives in city A and will return there, although this assumption simply reduces the problem of N cities to a problem of (N - 1) cities.

The path [A,D,C,B,E,A], with associated cost of 450 miles, is an example of a possible circuit. The goal description requires a complete circuit with minimum cost. Note

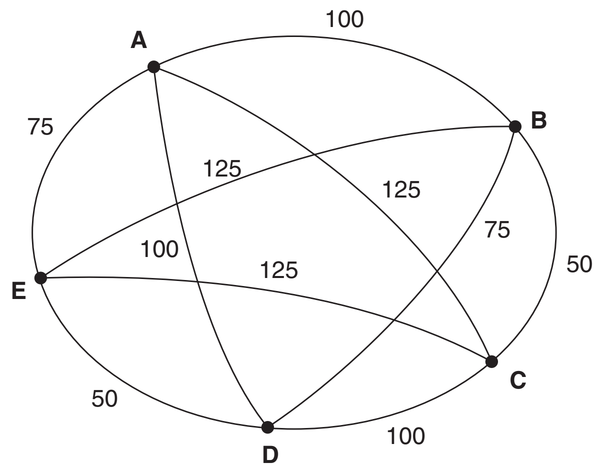



Figure 3.9 An instance of the traveling salesperson problem.

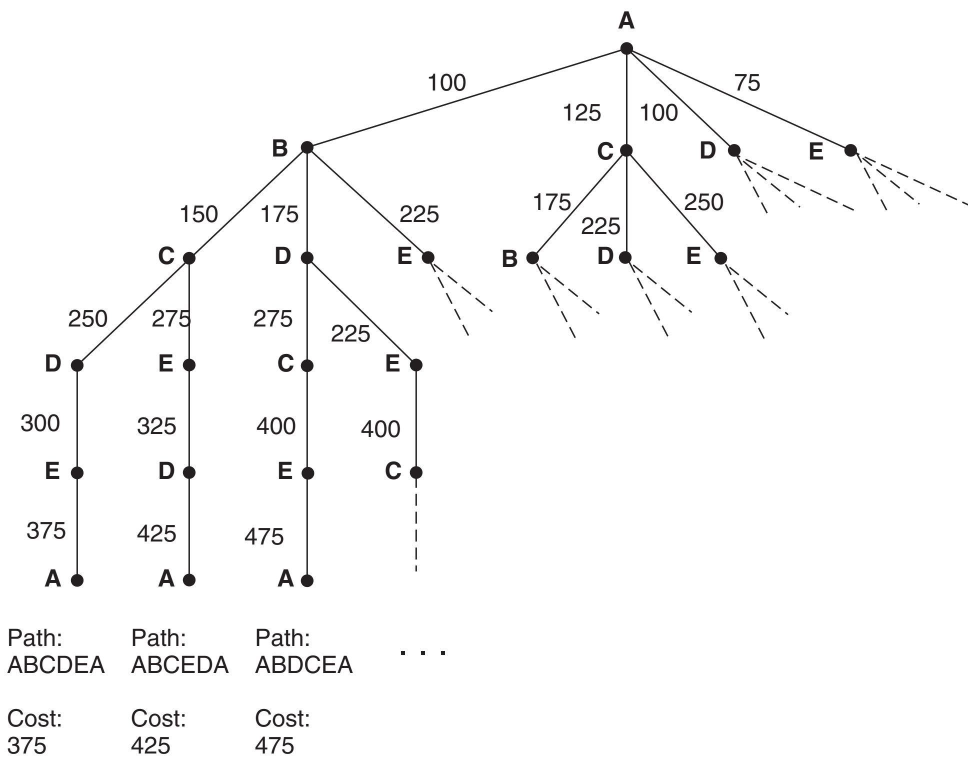



Figure 3.10 Search of the traveling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.

that the goal description is a property of the entire path, rather than of a single state. This is a goal description of type 2 from the definition of state space search.

Figure 3.10 shows one way in which possible solution paths may be generated and compared. Beginning with node A, possible next states are added until all cities are included and the path returns home. The goal is the lowest-cost path.

As Figure 3.10 suggests, the complexity of exhaustive search in the traveling salesperson problem is $(N - 1)!$, where N is the number of cities in the graph. For 9 cities we may exhaustively try all paths, but for any problem instance of interesting size, for example with 50 cities, simple exhaustive search cannot be performed within a practical length of time. In fact complexity costs for an $N!$ search grow so fast that very soon the search combinations become intractable.

Several techniques can reduce this search complexity. One is called *branch and bound* (Horowitz and Sahni 1978). Branch and bound generates paths one at a time, keeping track of the best circuit found so far. This value is used as a *bound* on future candidates. As paths are constructed one city at a time, the algorithm examines each partially completed path. If the algorithm determines that the best possible extension to a path, the branch, will have greater cost than the bound, it eliminates that partial path and *all* of its possible extensions. This reduces search considerably but still leaves an exponential number of paths ($1.26^N$ rather than $N!$).

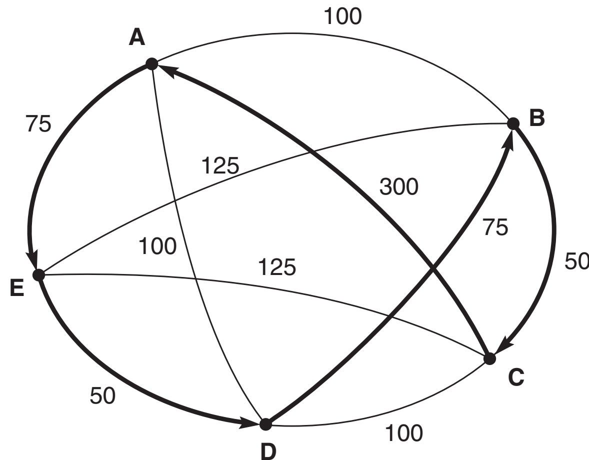



Figure 3.11 An instance of the traveling salesperson problem with the nearest neighbor path in bold. Note that this path (A, E, D, B, C, A), at a cost of 550, is not the shortest path. The comparatively high cost of arc (C, A) defeated the heuristic.

Another strategy for controlling search constructs the path according to the rule "go to the closest unvisited city." The *nearest neighbor* path through the graph of Figure 3.11 is [A,E,D,B,C,A], at a cost of 375 miles. This method is highly efficient, as there is only one path to be tried! The nearest neighbor, sometimes called *greedy*, heuristic is fallible, as graphs exist for which it does not find the shortest path, see Figure 3.11, but it is a possible compromise when the time required makes exhaustive search impractical.

Section 3.2 examines strategies for state space search.

## 3.2 Strategies for State Space Search

### 3.2.1 Data-Driven and Goal-Driven Search

A state space may be searched in two directions: from the given data of a problem instance toward a goal or from a goal back to the data.

In *data-driven search*, sometimes called *forward chaining*, the problem solver begins with the given facts of the problem and a set of legal moves or rules for changing state. Search proceeds by applying rules to facts to produce new facts, which are in turn used by the rules to generate more new facts. This process continues until (we hope!) it generates a path that satisfies the goal condition.

An alternative approach is possible: take the goal that we want to solve. See what rules or legal moves could be used to generate this goal and determine what conditions must be true to use them. These conditions become the new goals, or *subgoals*, for the search. Search continues, working backward through successive subgoals until (we hope!)

it works back to the facts of the problem. This finds the chain of moves or rules leading from data to a goal, although it does so in backward order. This approach is called *goal-driven* reasoning, or *backward chaining*, and it recalls the simple childhood trick of trying to solve a maze by working back from the finish to the start.

To summarize: data-driven reasoning takes the facts of the problem and applies the rules or legal moves to produce new facts that lead to a goal; goal-driven reasoning focuses on the goal, finds the rules that could produce the goal, and chains backward through successive rules and subgoals to the given facts of the problem.

In the final analysis, both data-driven and goal-driven problem solvers search the same state space graph; however, the order and actual number of states searched can differ. The preferred strategy is determined by the properties of the problem itself. These include the complexity of the rules, the "shape" of the state space, and the nature and availability of the problem data. All of these vary for different problems.

As an example of the effect a search strategy can have on the complexity of search, consider the problem of confirming or denying the statement "I am a descendant of Thomas Jefferson." A solution is a path of direct lineage between the "I" and Thomas Jefferson. This space may be searched in two directions, starting with the "I" and working along ancestor lines to Thomas Jefferson or starting with Thomas Jefferson and working through his descendants.

Some simple assumptions let us estimate the size of the space searched in each direction. Thomas Jefferson was born about 250 years ago; if we assume 25 years per generation, the required path will be about length 10. As each person has exactly two parents, a search back from the "I" would examine on the order of $2^{10}$ ancestors. A search that worked forward from Thomas Jefferson would examine more states, as people tend to have more than two children (particularly in the eighteenth and nineteenth centuries). If we assume an average of only three children per family, the search would examine on the order of $3^{10}$ nodes of the family tree. Thus, a search back from the "I" would examine fewer nodes. Note, however, that both directions yield exponential complexity.

The decision to choose between data- and goal-driven search is based on the structure of the problem to be solved. Goal-driven search is suggested if:

1. A goal or hypothesis is given in the problem statement or can easily be formulated. In a mathematics theorem prover, for example, the goal is the theorem to be proved. Many diagnostic systems consider potential diagnoses in a systematic fashion, confirming or eliminating them using goal-driven reasoning.

2. There are a large number of rules that match the facts of the problem and thus produce an increasing number of conclusions or goals. Early selection of a goal can eliminate most of these branches, making goal-driven search more effective in pruning the space (Figure 3.12). In a theorem prover, for example, the total number of rules used to produce a given theorem is usually much smaller than the number of rules that may be applied to the entire set of axioms.

3. Problem data are not given but must be acquired by the problem solver. In this case, goal-driven search can help guide data acquisition. In a medical diagnosis program, for example, a wide range of diagnostic tests can be applied. Doctors order only those that are necessary to confirm or deny a particular hypothesis.

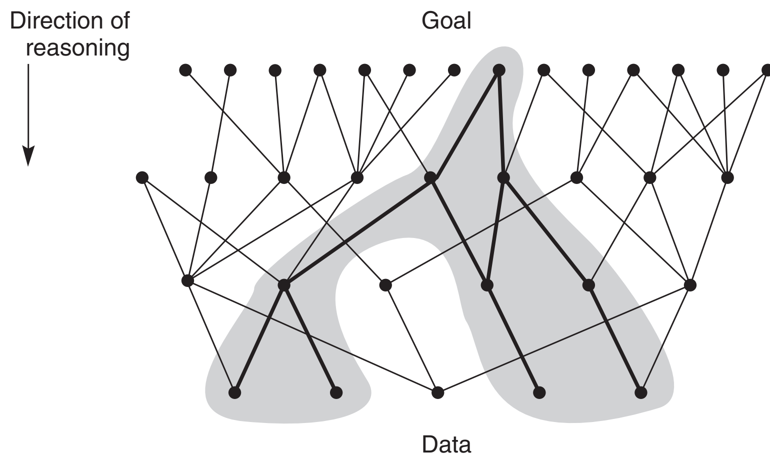



Figure 3.12 State space in which goal-directed search effectively prunes extraneous search paths.

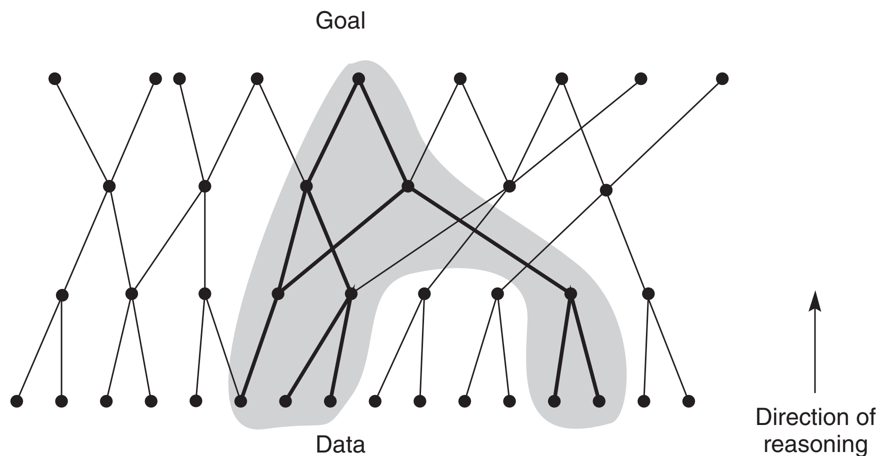



Figure 3.13 State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.

Goal-driven search thus uses knowledge of the desired goal to guide the search through relevant rules and eliminate branches of the space.

Data-driven search (Figure 3.13) is appropriate for problems in which:

1. All or most of the data are given in the initial problem statement. Interpretation problems often fit this mold by presenting a collection of data and asking the system to provide a high-level interpretation. Systems that analyze particular data (e.g., the PROSPECTOR or Dipmeter programs, which interpret geological data

or attempt to find what minerals are likely to be found at a site) fit the data-driven approach.

2. There are a large number of potential goals, but there are only a few ways to use the facts and given information of a particular problem instance. The DENDRAL program, an expert system that finds the molecular structure of organic compounds based on their formula, mass spectrographic data, and knowledge of chemistry, is an example of this. For any organic compound, there are an enormous number of possible structures. However, the mass spectrographic data on a compound allow DENDRAL to eliminate all but a few of these.
3. It is difficult to form a goal or hypothesis. In using DENDRAL, for example, little may be known initially about the possible structure of a compound.

Data-driven search uses the knowledge and constraints found in the given data of a problem to guide search along lines known to be true.

To summarize, there is no substitute for careful analysis of the particular problem to be solved, considering such issues as the *branching factor* of rule applications (see Chapter 4; on average, how many new states are generated by rule applications in both directions?), availability of data, and ease of determining potential goals.

### 3.2.2 Implementing Graph Search

In solving a problem using either goal- or data-driven search, a problem solver must find a path from a start state to a goal through the state space graph. The sequence of arcs in this path corresponds to the ordered steps of the solution. If a problem solver were given an oracle or other infallible mechanism for choosing a solution path, search would not be required. The problem solver would move unerringly through the space to the desired goal, constructing the path as it went. Because oracles do not exist for interesting problems, a problem solver must consider different paths through the space until it finds a goal. *Backtracking* is a technique for systematically trying all paths through a state space.

We begin with backtrack because it is one of the first search algorithms computer scientists study, and it has a natural implementation in a stack oriented recursive environment. We will present a simpler version of the backtrack algorithm with *depth-first search* (Section 3.2.3).

Backtracking search begins at the start state and pursues a path until it reaches either a goal or a "dead end." If it finds a goal, it quits and returns the solution path. If it reaches a dead end, it "backtracks" to the most recent node on the path having unexamined siblings and continues down one of these branches, as described in the following recursive rule:

> If the present state S does not meet the requirements of the goal description, then generate its first descendant $S_{child1}$, and apply the backtrack procedure recursively to this node. If backtrack does not find a goal node in the subgraph rooted at $S_{child1}$, repeat the procedure for its sibling, $S_{child2}$. Continue until either some descendant of a child is a goal node or all the children have been searched. If none of the children of S leads to a goal, then backtrack "fails back" to the parent of S, where it is applied to the siblings of S, and so on.

The algorithm continues until it finds a goal or exhausts the state space. Figure 3.14 shows the backtrack algorithm applied to a hypothetical state space. The direction of the dashed arrows on the tree indicates the progress of search up and down the space. The number beside each node indicates the order in which it is visited. We now define an algorithm that performs a backtrack, using three lists to keep track of nodes in the state space:

SL, for state list, lists the states in the current path being tried. If a goal is found, SL contains the ordered list of states on the solution path.

NSL, for new state list, contains nodes awaiting evaluation, i.e., nodes whose descendants have not yet been generated and searched.

DE, for dead ends, lists states whose descendants have failed to contain a goal. If these states are encountered again, they will be detected as elements of DE and eliminated from consideration immediately.

In defining the backtrack algorithm for the general case (a graph rather than a tree), it is necessary to detect multiple occurrences of any state so that it will not be reentered and cause (infinite) loops in the path. This is accomplished by testing each newly generated state for membership in any of these three lists. If a new state belongs to any of these lists, then it has already been visited and may be ignored.

```
function backtrack;
begin
  SL := [Start];  NSL := [Start];  DE := [ ];  CS := Start;                % initialize:
  while NSL ≠ [ ] do                               % while there are states to be tried
    begin
      if CS = goal (or meets goal description)
        then return SL;                   % on success, return list of states in path.
      if CS has no children (excluding nodes already on DE, SL, and NSL)
        then begin
          while SL is not empty and CS = the first element of SL do
            begin
              add CS to DE;                              % record state as dead end
              remove first element from SL;                              %backtrack
              remove first element from NSL;
              CS := first element of NSL;
            end
          add CS to SL;
        end
        else begin
          place children of CS (except nodes already on DE, SL, or NSL) on NSL;
          CS := first element of NSL;
          add CS to SL
        end
    end;
    return FAIL;
end.
```

In backtrack, the state currently under consideration is called CS for current state. CS is always equal to the state most recently added to SL and represents the "frontier" of the solution path currently being explored. Inference rules, moves in a game, or other appropriate problem-solving operators are ordered and applied to CS. The result is an ordered set of new states, the children of CS. The first of these children is made the new current state and the rest are placed in order on NSL for future examination. The new current state is added to SL and search continues. If CS has no children, it is removed from SL (this is where the algorithm "backtracks") and any remaining children of its predecessor on SL are examined.

A trace of backtrack on the graph of Figure 3.14 is given by:

Initialize: SL = [A]; NSL = [A]; DE = [ ]; CS = A;

| AFTER ITERATION | CS | SL | NSL | DE |
|---|---|---|---|---|
| 0 | A | [A] | [A] | [ ] |
| 1 | B | [B A] | [B C D A] | [ ] |
| 2 | E | [E B A] | [E F B C D A] | [ ] |
| 3 | H | [H E B A] | [H I E F B C D A] | [ ] |
| 4 | I | [I E B A] | [I E F B C D A] | [H] |
| 5 | F | [F B A] | [F B C D A] | [E I H] |
| 6 | J | [J F B A] | [J F B C D A] | [E I H] |
| 7 | C | [C A] | [C D A] | [B F J E I H] |
| 8 | G | [G C A] | [G C D A] | [B F J E I H] |

Figure 3.14 Backtracking search of a hypothetical state space.

As presented here, backtrack implements data-driven search, taking the root as a start state and evaluating its children to search for the goal. The algorithm can be viewed as a goal-driven search by letting the goal be the root of the graph and evaluating descendants back in an attempt to find a start state. If the goal description is of type 2 (see Section 3.1.3), the algorithm must determine a goal state by examining the path on SL.

backtrack is an algorithm for searching state space graphs. The graph search algorithms that follow, including depth-first, breadth-first, and best-first search, exploit the ideas used in backtrack, including:

1. The use of a list of unprocessed states (NSL) to allow the algorithm to return to any of these states.
2. A list of "bad" states (DE) to prevent the algorithm from retrying useless paths.
3. A list of nodes (SL) on the current solution path that is returned if a goal is found.
4. Explicit checks for membership of new states in these lists to prevent looping.

The next section introduces search algorithms that, like backtrack, use lists to keep track of states in a search space. These algorithms, including *depth-first*, *breadth-first*, and *best-first* (Chapter 4) search, differ from backtrack in providing a more flexible basis for implementing alternative graph search strategies.

### 3.2.3 Depth-First and Breadth-First Search

In addition to specifying a search direction (data-driven or goal-driven), a search algorithm must determine the order in which states are examined in the tree or the graph. This section considers two possibilities for the order in which the nodes of the graph are considered: *depth-first* and *breadth-first* search.

Consider the graph represented in Figure 3.15. States are labeled (A, B, C, . . .) so that they can be referred to in the discussion that follows. 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 into the search space whenever this is possible. Only when no further descendants of a state can be found are its siblings considered. Depth-first search examines the states in the graph of Figure 3.15 in the order A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R. The backtrack algorithm of Section 3.2.2 implemented depth-first search.

Breadth-first search, in contrast, explores the space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next deeper level. A breadth-first search of the graph of Figure 3.15 considers the states in the order A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.

We implement breadth-first search using lists, open and closed, to keep track of progress through the state space. open, like NSL in backtrack, lists states that have been generated but whose children have not been examined. The order in which states are removed from open determines the order of the search. closed records states already examined. closed is the union of the DE and SL lists of the backtrack algorithm.

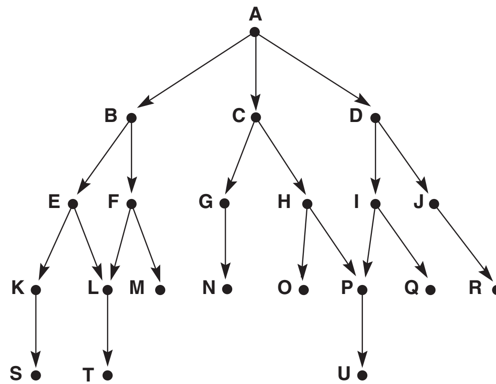



Figure 3.15 Graph for breadth- and depth-first search examples.

```
function breadth_first_search;

begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                         % goal found
               else begin
                  generate children of X;
                  put X on closed;
                  discard children of X if already on open or closed;  % loop check
                  put remaining children on right end of open          % queue
               end
      end
   return FAIL                                                         % no states left
end.
```

Child states are generated by inference rules, legal moves of a game, or other state transition operators. Each iteration produces all children of the state X and adds them to open. Note that open is maintained as a *queue*, or first-in-first-out (FIFO) data structure. States are added to the right of the list and removed from the left. This biases search toward the states that have been on open the longest, causing the search to be breadth-first. Child states that have already been discovered (already appear on either open or closed) are discarded. If the algorithm terminates because the condition of the "while" loop is no longer satisfied (open = [ ]) then it has searched the entire graph without finding the desired goal: the search has failed.

A trace of breadth_first_search on the graph of Figure 3.15 follows. Each successive number, 2,3,4, . . . , represents an iteration of the "while" loop. U is the goal state.

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] (as L is already on open); 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 = [ ].

Figure 3.16 illustrates the graph of Figure 3.15 after six iterations of breadth_first_search. The states on open and closed are highlighted. States not shaded have not been discovered by the algorithm. Note that open records the states on the "frontier" of the search at any stage and that closed records states already visited.

Because breadth-first search considers every node at each level of the graph before going deeper into the space, all states are first reached along the shortest path from the start state. Breadth-first search is therefore guaranteed to find the shortest path from the start state to the goal. Furthermore, because all states are first found along the shortest path, any states encountered a second time are found along a path of equal or greater length. Because there is no chance that duplicate states were found along a better path, the algorithm simply discards any duplicate states.

It is often useful to keep other information on open and closed besides the names of the states. For example, note that breadth_first_search does not maintain a list of states on the current path to a goal as backtrack did on the list SL; all visited states are kept on closed. If a solution path is required, it can not be returned by this algorithm. The solution can be found by storing ancestor information along with each state. A state may be saved along with a record of its parent state, e.g., as a (state, parent) pair. If this is done in the search of Figure 3.15, the contents of open and closed at the fourth iteration would be:

open = [(D,A), (E,B), (F,B), (G,C), (H,C)]; closed = [(C,A), (B,A), (A,nil)]

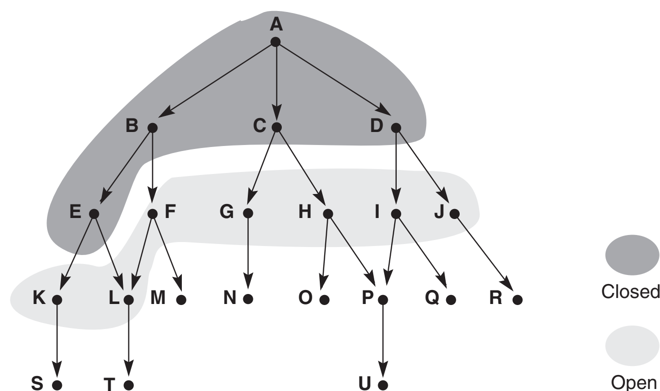



Figure 3.16 Graph of Figure 3.15 at iteration 6 of breadth-first search. States on open and closed are highlighted.

The path (A, B, F) that led from A to F could easily be constructed from this information. When a goal is found, the algorithm can construct the solution path by tracing back along parents from the goal to the start state. Note that state A has a parent of nil, indicating that it is a start state; this stops reconstruction of the path. Because breadth-first search finds each state along the shortest path and retains the first version of each state, this is the shortest path from a start to a goal.

Figure 3.17 shows the states removed from open and examined in a breadth-first search of the graph of the 8-puzzle. As before, arcs correspond to moves of the blank up, to the right, down, and to the left. The number next to each state indicates the order in which it was removed from open. States left on open when the algorithm halted are not shown.

Next, we create a depth-first search algorithm, a simplification of the backtrack algorithm already presented in Section 3.2.3. In this algorithm, the descendant states are added and removed from the *left* end of open: open is maintained as a *stack*, or last-in-first-out (LIFO) structure. The organization of open as a stack directs search toward the most recently generated states, producing a depth-first search order:

```
function depth_first_search;

begin
  open := [Start];                                                % initialize
  closed := [ ];
  while open ≠ [ ] do                                             % states remain
    begin
      remove leftmost state from open, call it X;
      if X is a goal then return SUCCESS                          % goal found
        else begin
          generate children of X;
          put X on closed;
          discard children of X if already on open or closed;     % loop check
          put remaining children on left end of open              % stack
        end
    end;
  return FAIL                                                     % no states left
end.
```

A trace of depth_first_search on the graph of Figure 3.15 appears below. Each successive iteration of the "while" loop is indicated by a single line (2, 3, 4, . . .). The initial states of open and closed are given on line 1. Assume U is the goal state.

1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]

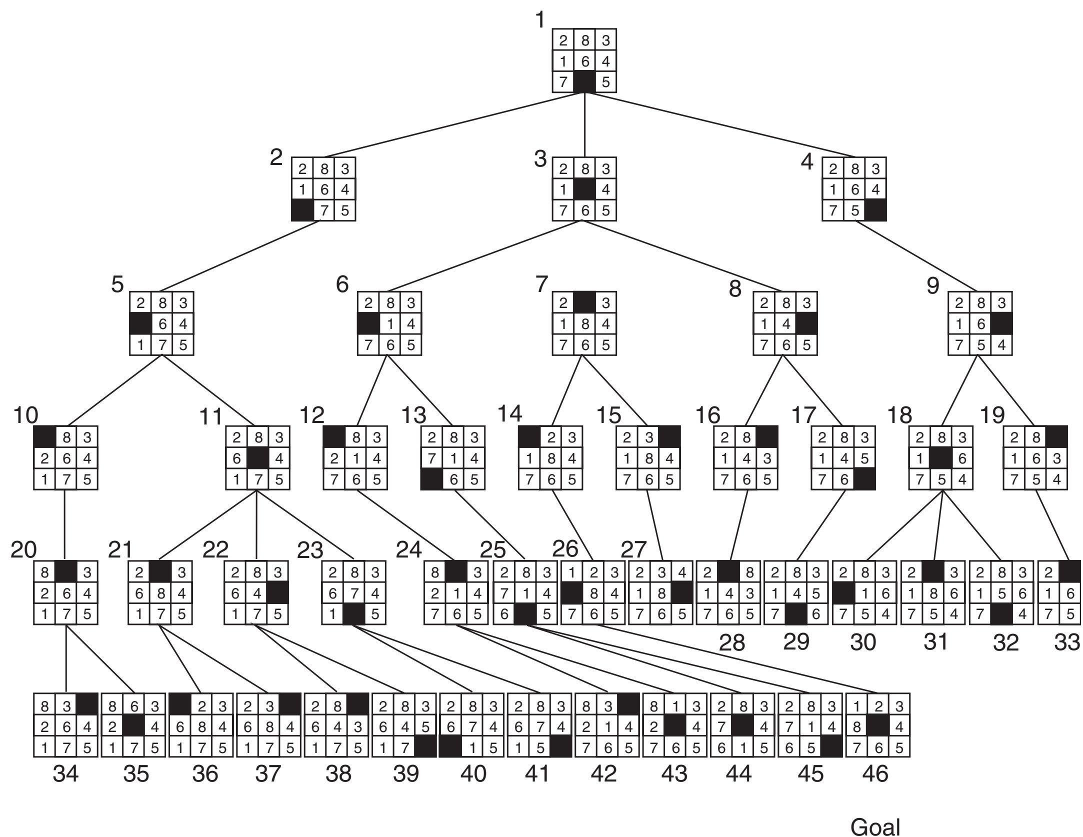



Figure 3.17 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]
9. open = [M,C,D], (as L is already on closed); closed = [F,T,L,S,K,E,B,A]
10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]

and so on until either U is discovered or open = [ ].

As with breadth_first_search, open lists all states discovered but not yet evaluated (the current "frontier" of the search), and closed records states already considered. Figure 3.18 shows the graph of Figure 3.15 at the sixth iteration of the depth_first_search. The contents of open and closed are highlighted. As with breadth_first_search, the algorithm could store a record of the parent along with each state, allowing the algorithm to reconstruct the path that led from the start state to a goal.

Unlike breadth-first search, a depth-first search is not guaranteed to find the shortest path to a state the first time that state is encountered. Later in the search, a different path may be found to any state. If path length matters in a problem solver, when the algorithm encounters a duplicate state, the algorithm should save the version reached along the shortest path. This could be done by storing each state as a triple: (state, parent,

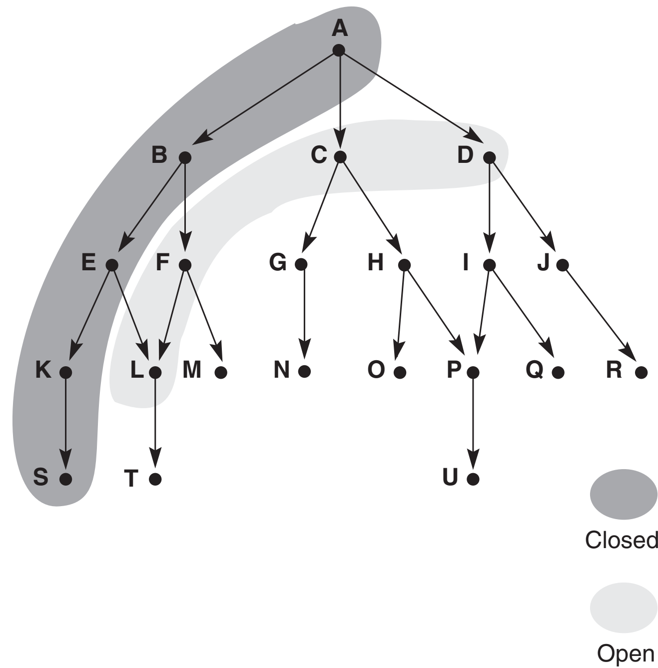



Figure 3.18 Graph of Figure 3.15 at iteration 6 of depth-first search. States on open and closed are highlighted.

length_of_path). When children are generated, the value of the path length is simply incremented by one and saved with the child. If a child is reached along multiple paths, this information can be used to retain the best version. This is treated in more detail in the discussion of *algorithm A* in Chapter 4. Note that retaining the best version of a state in a simple depth-first search does not guarantee that a goal will be reached along the shortest path.

Figure 3.19 gives a depth-first search of the 8-puzzle. As noted previously, the space is generated by the four "move blank" rules (up, down, left, and right). The numbers next to the states indicate the order in which they were considered, i.e., removed from open. States left on open when the goal is found are not shown. A depth bound of 5 was imposed on this search to keep it from getting lost deep in the space.

As with choosing between data- and goal-driven search for evaluating a graph, the choice of depth-first or breadth-first search depends on the specific problem being solved. Significant features include the importance of finding the shortest path to a goal, the branching factor of the space, the available compute time and space resources, the average length of paths to a goal node, and whether we want all solutions or only the first solution. In making these decisions, there are advantages and disadvantages for each approach.

**Breadth-First** Because it always examines all the nodes at level n before proceeding to level n + 1, breadth-first search always finds the shortest path to a goal node. In a problem where it is known that a simple solution exists, this solution will be found. Unfortunately, if there is a bad branching factor, i.e., states have a high average number of children, the

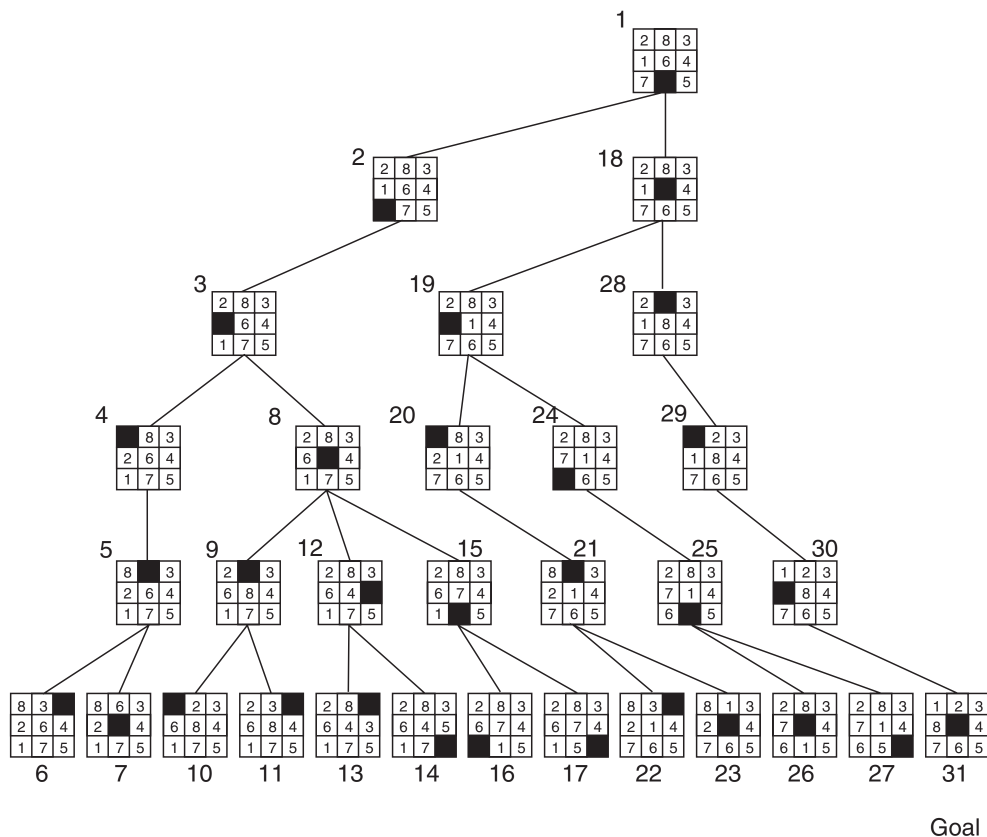



Figure 3.19 Depth-first search of the 8-puzzle with a depth bound of 5.

combinatorial explosion may prevent the algorithm from finding a solution using available memory. This is due to the fact that all unexpanded nodes for each level of the search must be kept on open. For deep searches, or state spaces with a high branching factor, this can become quite cumbersome.

The space utilization of breadth-first search, measured in terms of the number of states on open, is an exponential function of the length of the path at any time. If each state has an average of B children, the number of states on a given level is B times the number of states on the previous level. This gives $B^n$ states on level n. Breadth-first search would place all of these on open when it begins examining level n. This can be prohibitive if solution paths are long, in the game of chess, for example.

**Depth-First** Depth-first search gets quickly into a deep search space. If it is known that the solution path will be long, depth-first search will not waste time searching a large number of "shallow" states in the graph. On the other hand, depth-first search can get "lost" deep in a graph, missing shorter paths to a goal or even becoming stuck in an infinitely long path that does not lead to a goal.

Depth-first search is much more efficient for search spaces with many branches because it does not have to keep all the nodes at a given level on the open list. The space

usage of depth-first search is a linear function of the length of the path. At each level, open retains only the children of a single state. If a graph has an average of B children per state, this requires a total space usage of $B \times n$ states to go n levels deep into the space.

The best answer to the "depth-first versus breadth-first" issue is to examine the problem space and consult experts in the area. In chess, for example, breadth-first search simply is not possible. In simpler games, breadth-first search not only may be possible but, because it gives the shortest path, may be the only way to avoid losing.

### 3.2.4 Depth-First Search with Iterative Deepening

A nice compromise on these trade-offs is to use a depth bound on depth-first search. The depth bound forces a failure on a search path once it gets below a certain level. This causes a breadth-first like sweep of the search space at that depth level. When it is known that a solution lies within a certain depth or when time constraints, such as those that occur in an extremely large space like chess, limit the number of states that can be considered; then a depth-first search with a depth bound may be most appropriate. Figure 3.19 showed a depth-first search of the 8-puzzle in which a depth bound of 5 caused the sweep across the space at that depth.

This insight leads to a search algorithm that remedies many of the drawbacks of both depth-first and breadth-first search. *Depth-first iterative deepening* (Korf 1987) performs a depth-first search of the space with a depth bound of 1. If it fails to find a goal, it performs another depth-first search with a depth bound of 2. This continues, increasing the depth bound by one each time. At each iteration, the algorithm performs a complete depth-first search to the current depth bound. No information about the state space is retained between iterations.

Because the algorithm searches the space in a level-by-level fashion, it is guaranteed to find a shortest path to a goal. Because it does only depth-first search at each iteration, the space usage at any level n is $B \times n$, where B is the average number of children of a node.

Interestingly, although it seems as if depth-first iterative deepening would be much less efficient than either depth-first or breadth-first search, its time complexity is actually of the same order of magnitude as either of these: $O(B^n)$. An intuitive explanation for this seeming paradox is given by Korf (1987):

> Since the number of nodes in a given level of the tree grows exponentially with depth, almost all the time is spent in the deepest level, even though shallower levels are generated an arithmetically increasing number of times.

Unfortunately, all the search strategies discussed in this chapter—depth-first, breadth-first, and depth-first iterative deepening—may be shown to have worst-case exponential time complexity. This is true for all *uninformed* search algorithms. The only approaches to search that reduce this complexity employ heuristics to guide search. *Best-first search* is a search algorithm that is similar to the algorithms for depth- and breadth-first search just presented. However, best-first search orders the states on the

open list, the current fringe of the search, according to some measure of their heuristic merit. At each iteration, it considers neither the deepest nor the shallowest but the "best" state. Best-first search is the main topic of Chapter 4.

## 3.3 Using the State Space to Represent Reasoning with the Propositional and Predicate Calculus

### 3.3.1 State Space Description of a Logic System

When we defined state space graphs in Section 3.1, we noted that nodes must be distinguishable from one another, with each node representing some state of the solution process. The propositional and predicate calculus can be used as the formal specification language for making these distinctions as well as for mapping the nodes of a graph onto the state space. Furthermore, inference rules can be used to create and describe the arcs between states. In this fashion, problems in the predicate calculus, such as determining whether a particular expression is a logical consequence of a given set of assertions, may be solved using search.

The soundness and completeness of predicate calculus inference rules can guarantee the correctness of conclusions derived through this form of graph-based reasoning. This ability to produce a formal proof of the integrity of a solution through the same algorithm that produces the solution is a unique attribute of much artificial intelligence and theorem proving based problem solving.

Although many problems' states, e.g., tic-tac-toe, can be more naturally described by other data structures, such as arrays, the power and generality of logic allow much of AI problem solving to use the propositional and predicate calculus descriptions and inference rules. Other AI representations such as rules (Chapter 8), semantic networks, or frames (Chapter 7) also employ search strategies similar to those presented in Section 3.2.

#### EXAMPLE 3.3.1: THE PROPOSITIONAL CALCULUS

The first example of how a set of logic relationships may be viewed as defining a graph is from the propositional calculus. If p, q, r,... are propositions, assume the assertions:

$q \rightarrow p$
$r \rightarrow p$
$v \rightarrow q$
$s \rightarrow r$
$t \rightarrow r$
$s \rightarrow u$
$s$
$t$

# 4 HEURISTIC SEARCH

*The task that a symbol system is faced with, then, when it is presented with a problem and a problem space, is to use its limited processing resources to generate possible solutions, one after another, until it finds one that satisfies the problem defining test. If the symbol system had some control over the order in which potential solutions were generated, then it would be desirable to arrange this order of generation so that actual solutions would have a high likelihood of appearing early. A symbol system would exhibit intelligence to the extent that it succeeded in doing this. Intelligence for a system with limited processing resources consists in making wise choices of what to do next. . . .*

—NEWELL AND SIMON, 1976, Turing Award Lecture

*I been searchin' . . .*
*Searchin' . . . Oh yeah,*
*Searchin' every which-a-way . . .*

—LIEBER AND STOLLER

## 4.0 Introduction

George Polya defines *heuristic* as "the study of the methods and rules of discovery and invention" (Polya 1945). This meaning can be traced to the term's Greek root, the verb *eurisco*, which means "I discover." When Archimedes emerged from his famous bath clutching the golden crown, he shouted "Eureka!" meaning "I have found it!". In state space search, *heuristics* are formalized as rules for choosing those branches in a state space that are most likely to lead to an acceptable problem solution.

AI problem solvers employ heuristics in two basic situations:

1. A problem may not have an exact solution because of inherent ambiguities in the problem statement or available data. Medical diagnosis is an example of this. A given set of symptoms may have several possible causes; doctors use heuristics

to choose the most likely diagnosis and formulate a plan of treatment. Vision is another example of an inexact problem. Visual scenes are often ambiguous, allowing multiple interpretations of the connectedness, extent, and orientation of objects. Optical illusions exemplify these ambiguities. Vision systems often use heuristics to select the most likely of several possible interpretations of as scene.

2. A problem may have an exact solution, but the computational cost of finding it may be prohibitive. In many problems (such as chess), state space growth is combinatorially explosive, with the number of possible states increasing exponentially or factorially with the depth of the search. In these cases, exhaustive, *brute-force* search techniques such as depth-first or breadth-first search may fail to find a solution within any practical length of time. Heuristics attack this complexity by guiding the search along the most "promising" path through the space. By eliminating unpromising states and their descendants from consideration, a heuristic algorithm can (its designer hopes) defeat this *combinatorial explosion* and find an acceptable solution.

Unfortunately, like all rules of discovery and invention, heuristics are fallible. A heuristic is only an informed guess of the next step to be taken in solving a problem. It is often based on experience or intuition. Because heuristics use limited information, such as knowledge of the present situation or descriptions of states currently on the open list, they are not always able to predict the exact behavior of the state space farther along in the search. A heuristic can lead a search algorithm to a suboptimal solution or fail to find any solution at all. This is an inherent limitation of heuristic search. It cannot be eliminated by "better" heuristics or more efficient search algorithms (Garey and Johnson 1979).

Heuristics and the design of algorithms to implement heuristic search have long been a core concern of artificial intelligence. Game playing and theorem proving are two of the oldest applications in artificial intelligence; both of these require heuristics to prune spaces of possible solutions. It is not feasible to examine every inference that can be made in a mathematics domain or every possible move that can be made on a chessboard. Heuristic search is often the only practical answer.

Expert systems research has affirmed the importance of heuristics as an essential component of problem solving. When a human expert solves a problem, he or she examines the available information and makes a decision. The "rules of thumb" that a human expert uses to solve problems efficiently are largely heuristic in nature. These heuristics are extracted and formalized by expert systems designers, as we see in Chapter 8.

It is useful to think of heuristic search from two perspectives: the heuristic measure and an algorithm that uses heuristics to search the state space. In Section 4.1, we implement heuristics with *hill-climbing* and *dynamic programming* algorithms. In Section 4.2 we present an algorithm for *best-first* search. The design and evaluation of the effectiveness of heuristics is presented in Section 4.3, and game playing heuristics in Section 4.4.

Consider heuristics in the game of tic-tac-toe, Figure II.5. The combinatorics for exhaustive search are high but not insurmountable. Each of the nine first moves has eight possible responses, which in turn have seven continuing moves, and so on. A simple analysis puts the total number of states for exhaustive search at $9 \times 8 \times 7 \times \cdots$ or 9!.

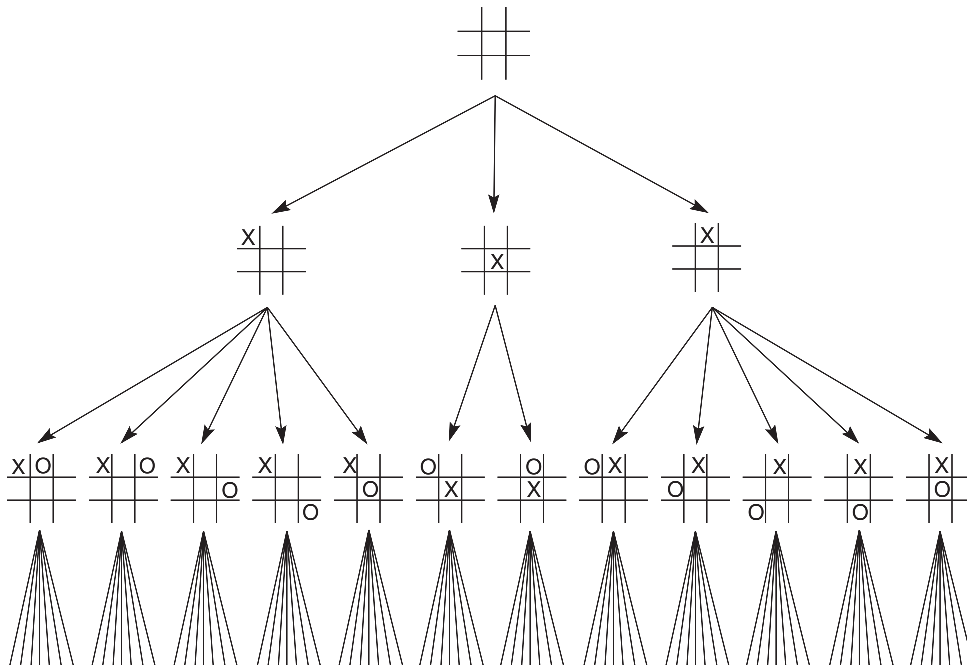



Figure 4.1 First three levels of the tic-tac-toe state space reduced by symmetry.

Symmetry reduction decreases the search space. Many problem configurations are actually equivalent under symmetric operations of the game board. Thus, there are not nine but really three initial moves: to a corner, to the center of a side, and to the center of the grid. Use of symmetry on the second level further reduces the number of paths through the space to $12 \times 7!$, as seen in Figure 4.1. Symmetries in a game space such as this may be described as mathematical invariants, that, when they exist, can often be used to tremendous advantage in reducing search.

A simple heuristic, however, can almost eliminate search entirely: we may move to the state in which X has the most winning opportunities. (The first three states in the tic-tac-toe game are so measured in Figure 4.2.) In case of states with equal numbers of potential wins, take the first such state found. The algorithm then selects and moves to the state with the highest number of opportunities. In this case X takes the center of the grid. Note that not only are the other two alternatives eliminated, but so are all their descendants. Two-thirds of the full space is pruned away with the first move, Figure 4.3.

After the first move, the opponent can choose either of two alternative moves (as seen in Figure 4.3). Whichever is chosen, the heuristic can be applied to the resulting state of the game, again using "most winning opportunities" to select among the possible moves. As search continues, each move evaluates the children of a single node; exhaustive search is not required. Figure 4.3 shows the reduced search after three steps in the game. States are marked with their heuristic values. Although not an exact calculation of search size for

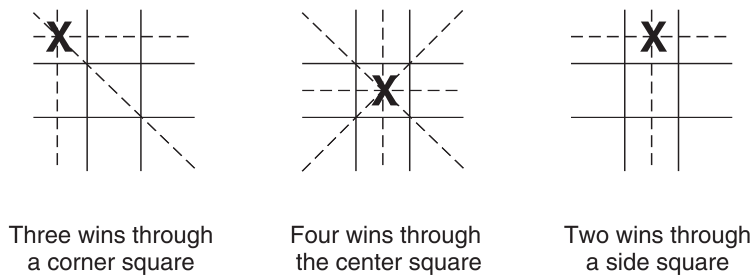



Figure 4.2 The most wins heuristic applied to the first children in tic-tac-toe.

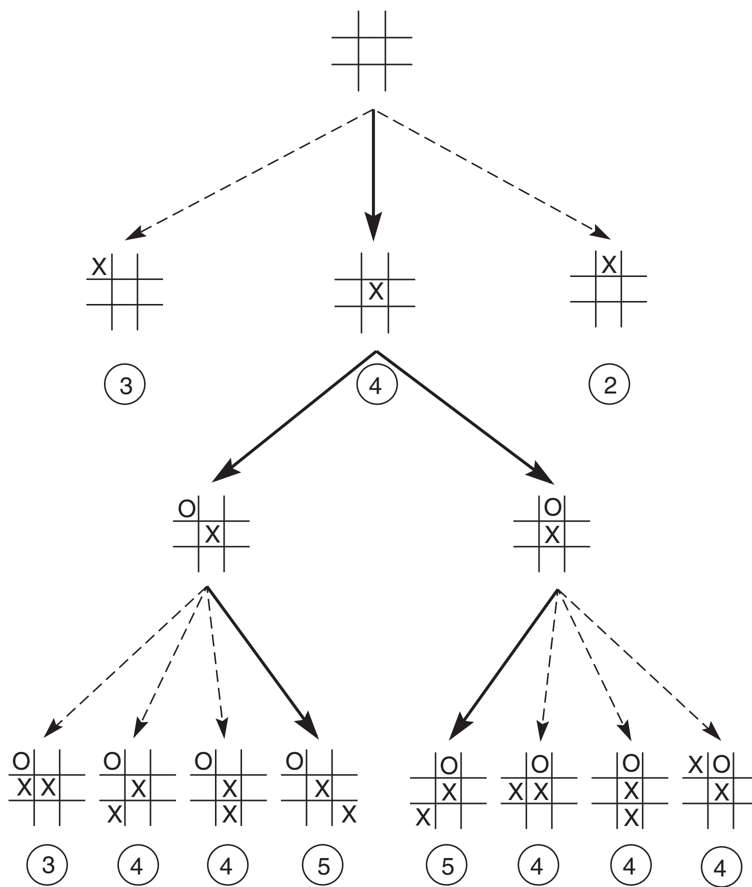



Figure 4.3 Heuristically reduced state space for tic-tac-toe.

this “most wins” strategy for tic-tac-toe, a crude upper bound can be computed by assuming a maximum of five moves in a game with five options per move. In reality, the number of states is smaller, as the board fills and reduces options. This crude bound of 25 states is an improvement of four orders of magnitude over 9!.

The next section presents two algorithms for implementing heuristic search: *hill-climbing* and *dynamic programming*. Section 4.2 uses the priority queue for *best-first* search. In Section 4.3 we discuss theoretical issues related to heuristic search, such as *admissibility* and *monotonicity.* Section 4.4 examines the use of *minimax* and *alpha-beta pruning* to apply heuristics to two-person games. The final section of Chapter 4 examines the complexity of heuristic search and reemphasizes its essential role in intelligent problem solving.

## 4.1 Hill-Climbing and Dynamic Programming

### 4.1.1 Hill-Climbing

The simplest way to implement heuristic search is through a procedure called *hill-climbing* (Pearl 1984). Hill-climbing strategies expand the current state of the search and evaluate its children. The "best" child is selected for further expansion; neither its siblings nor its parent are retained. Hill climbing is named for the strategy that might be used by an eager, but blind mountain climber: go uphill along the steepest possible path until you can go no farther up. Because it keeps no history, the algorithm cannot recover from failures of its strategy. An example of hill-climbing in tic-tac-toe was the "take the state with the most possible wins" that we demonstrated in Section 4.0.

A major problem of hill-climbing strategies is their tendency to become stuck at *local maxima*. If they reach a state that has a better evaluation than any of its children, the algorithm falters. If this state is not a goal, but just a local maximum, the algorithm may fail to find the best solution. That is, performance might well improve in a limited setting, but because of the shape of the entire space, it may never reach the overall best. An example of local maxima in games occurs in the 8-puzzle. Often, in order to move a particular tile to its destination, other tiles already in goal position need be moved out. This is necessary to solve the puzzle but temporarily worsens the board state. Because "better" need not be "best" in an absolute sense, search methods without backtracking or some other recovery mechanism are unable to distinguish between local and global maxima.

Figure 4.4 is an example of the local maximum dilemma. Suppose, exploring this search space, we arrive at state X, wanting to maximize state values. The evaluations of X's children, grand children, and great grandchildren demonstrate that hill-climbing can get confused even with multiple level look ahead. There are methods for getting around this problem, such as randomly perturbing the evaluation function, but in general there is no way of guaranteeing optimal performance with hill-climbing techniques. Samuel's (1959) checker program offers an interesting variant of the hill climbing algorithm.

Samuel's program was exceptional for its time, 1959, particularly given the limitations of the 1950s computers. Not only did Samuel's program apply heuristic search to checker playing, but it also implemented algorithms for optimal use of limited memory, as well as a simple form of learning. Indeed, it pioneered many of the techniques still used in game-playing and machine learning programs.

Samuel's program evaluated board states with a weighted sum of several different heuristic measures:

|  | — | e | x | e | c | u | t | i | o | n |
|---|---|---|---|---|---|---|---|---|---|---|
| — | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| i | 1 | **2** | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| n | 2 | 3 | **4** | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| t | 3 | 4 | **5** | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| e | 4 | 5 | 6 | **5** | **6** | 7 | 8 | 9 | 10 | 11 |
| n | 5 | 6 | 7 | 6 | 7 | **8** | 9 | 10 | 11 | 12 |
| t | 6 | 7 | 8 | 7 | 8 | 9 | **8** | 9 | 10 | 11 |
| i | 7 | 8 | 9 | 8 | 9 | 10 | 9 | **8** | 9 | 10 |
| o | 8 | 9 | 10 | 9 | 10 | 11 | 10 | 9 | **8** | 9 |
| n | 9 | 10 | 11 | 10 | 11 | 12 | 11 | 10 | 9 | **8** |

Figure 4.9 Complete array of minimum edit difference between intention and execution (adapted from Jurafsky and Martin 2000).

In the spell check situation of proposing a cost-based ordered list of words for replacing an unrecognized string, the backward segment of the dynamic programming algorithm is not needed. Once the minimum edit measure is calculated for the set of related strings a prioritized order of alternatives is proposed from which the user chooses an appropriate string.

The justification for dynamic programming is the cost of time/space in computation. Dynamic programming, as seen in our examples has cost of $n^2$, where $n$ is the length of the largest string; the cost in the worse case is $n^3$, if other related subproblems need to be considered (other row/column values) to determine the current state. Exhaustive search for comparing two strings is exponential, costing between $2^n$ and $3^n$.

There are a number of obvious heuristics that can be used to prune the search in dynamic programming. First, useful solutions will usually lie around the upper left to lower right diagonal of the array; this leads to ignoring development of array extremes. Second, it can be useful to prune the search as it evolves, e.g., for edit distances passing a certain threshold, cut that solution path or even abandon the whole problem, i.e., the source string will be so distant from the target string of characters as to be useless. There is also a stochastic approach to the pattern comparison problem that we will see in Section 5.3.

## 4.2 The Best-First Search Algorithm

### 4.2.1 Implementing Best-First Search

In spite of their limitations, algorithms such as backtrack, hill climbing, and dynamic programming can be used effectively if their evaluation functions are sufficiently informative to avoid local maxima, dead ends, and related anomalies in a search space. In general,

however, use of heuristic search requires a more flexible algorithm: this is provided by *best-first search*, where, with a priority queue, recovery from these situations is possible.

Like the depth-first and breadth-first search algorithms of Chapter 3, best-first search uses lists to maintain states: open to keep track of the current fringe of the search and closed to record states already visited. An added step in the algorithm orders the states on open according to some heuristic estimate of their "closeness" to a goal. Thus, each iteration of the loop considers the most "promising" state on the open list. The pseudo-code for the function best_first_search appears below.

```
function best_first_search;

begin
  open := [Start];                                                % initialize
  closed := [ ];
  while open ≠ [ ] do                                         % states remain
    begin
      remove the leftmost state from open, call it X;
      if X = goal then return the path from Start to X
      else begin
            generate children of X;
            for each child of X do
            case
                the child is not on open or closed:
                  begin
                      assign the child a heuristic value;
                      add the child to open
                  end;
                the child is already on open:
                  if the child was reached by a shorter path
                  then give the state on open the shorter path
                the child is already on closed:
                  if the child was reached by a shorter path then
                    begin
                      remove the state from closed;
                      add the child to open
                    end;
            end;                                                      % case
            put X on closed;
            re-order states on open by heuristic merit (best leftmost)
          end;
  return FAIL                                                % open is empty
end.
```

At each iteration, best_first_search removes the first element from the open list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Note that each state retains ancestor information to determine if it had previously been reached by a shorter path and to allow the algorithm to return the final solution path. (See Section 3.2.3.)

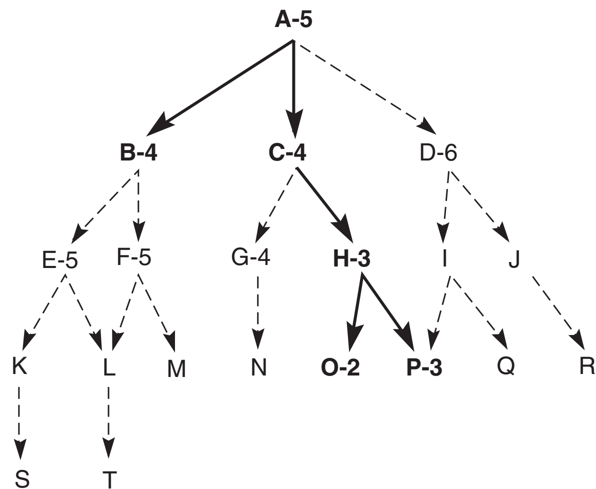



Figure 4.10 Heuristic search of a hypothetical state space.

If the first element on open is not a goal, the algorithm applies all matching production rules or operators to generate its descendants. If a child state is not on open or closed best_first_search applies a heuristic evaluation to that state, and the open list is sorted according to the heuristic values of those states. This brings the "best" states to the front of open. Note that because these estimates are heuristic in nature, the next "best" state to be examined may be from any level of the state space. When open is maintained as a sorted list, it is often referred to as a *priority queue*.

If a child state is already on open or closed, the algorithm checks to make sure that the state records the shorter of the two partial solution paths. Duplicate states are not retained. By updating the path history of nodes on open and closed when they are rediscovered, the algorithm will find a shortest path to a goal (within the states considered).

Figure 4.10 shows a hypothetical state space with heuristic evaluations attached to some of its states. The states with attached evaluations are those actually generated in best_first_search. The states expanded by the heuristic search algorithm are indicated in bold; note that it does not search all of the space. The goal of best-first search is to find the goal state by looking at as few states as possible; the more *informed* (Section 4.2.3) the heuristic, the fewer states are processed in finding the goal.

A trace of the execution of best_first_search on this graph appears below. Suppose P is the goal state in the graph of Figure 4.10. Because P is the goal, states along the path to P will tend to have lower heuristic values. The heuristic is fallible: the state O has a lower value than the goal itself and is examined first. Unlike hill climbing, which does not maintain a priority queue for the selection of "next" states, the algorithm recovers from this error and finds the correct goal.

1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]

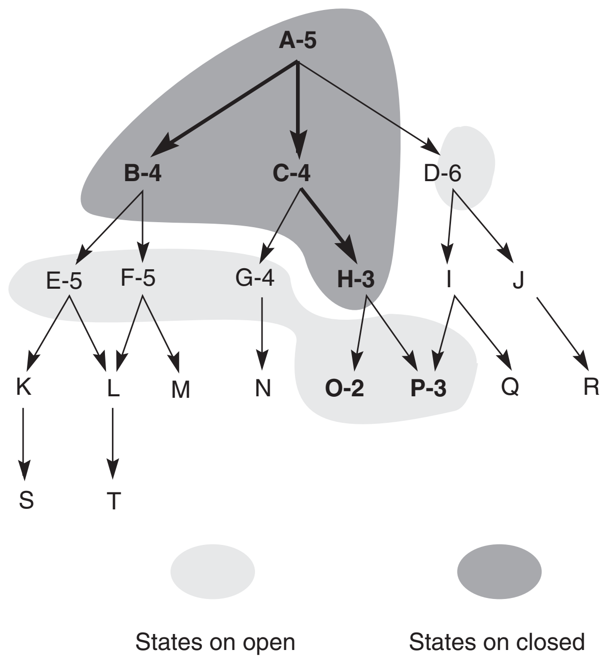



Figure 4.11 Heuristic search of a hypothetical state space with open and closed states highlighted.

3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!

Figure 4.11 shows the space as it appears after the fifth iteration of the while loop. The states contained in open and closed are indicated. open records the current frontier of the search and closed records states already considered. Note that the frontier of the search is highly uneven, reflecting the opportunistic nature of best-first search.

The best-first search algorithm selects the most promising state on open for further expansion. However, as it is using a heuristic that may prove erroneous, it does not abandon all the other states but maintains them on open. In the event a heuristic leads the search down a path that proves incorrect, the algorithm will eventually retrieve some previously generated, “next best” state from open and shift its focus to another part of the space. In the example of Figure 4.10, after the children of state B were found to have poor heuristic evaluations, the search shifted its focus to state C. The children of B were kept on open in case the algorithm needed to return to them later. In best_first_search, as in the algorithms of Chapter 3, the open list supports backtracking from paths that fail to produce a goal.

### 4.2.2 Implementing Heuristic Evaluation Functions

We next evaluate the performance of several different heuristics for solving the 8-puzzle. Figure 4.12 shows a start and goal state for the 8-puzzle, along with the first three states generated in the search.

The simplest heuristic counts the tiles out of place in each state when compared with the goal. This is intuitively appealing, because it would seem that, all else being equal, the state that had fewest tiles out of place is probably closer to the desired goal and would be the best to examine next.

However, this heuristic does not use all of the information available in a board configuration, because it does not take into account the distance the tiles must be moved. A "better" heuristic would sum all the distances by which the tiles are out of place, one for each square a tile must be moved to reach its position in the goal state.

Both of these heuristics can be criticized for failing to acknowledge the difficulty of tile reversals. That is, if two tiles are next to each other and the goal requires their being in opposite locations, it takes (several) more than two moves to put them back in place, as the tiles must "go around" each other (Figure 4.13).

A heuristic that takes this into account multiplies a small number (2, for example) times each direct tile reversal (where two adjacent tiles must be exchanged to be in the order of the goal). Figure 4.14 shows the result of applying each of these three heuristics to the three child states of Figure 4.12.

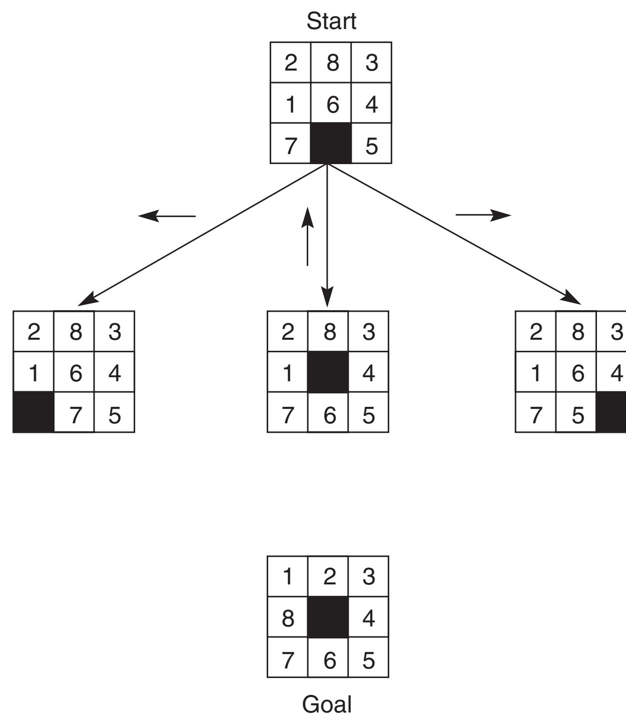



Figure 4.12 The start state, first moves, and goal state for an example 8-puzzle.

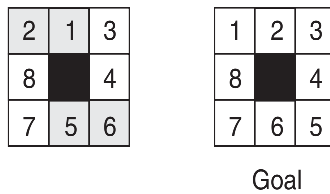



Figure 4.13 An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.

In Figure 4.14's summary of evaluation functions, the sum of distances heuristic does indeed seem to provide a more accurate estimate of the work to be done than the simple count of the number of tiles out of place. Also, the tile reversal heuristic fails to distinguish between these states, giving each an evaluation of 0. Although it is an intuitively appealing heuristic, it breaks down since none of these states have any direct reversals. A fourth heuristic, which may overcome the limitations of the tile reversal heuristic, adds the sum of the distances the tiles are out of place and 2 times the number of direct reversals.

This example illustrates the difficulty of devising good heuristics. Our goal is to use the limited information available in a single state descriptor to make intelligent choices. Each of the heuristics proposed above ignores some critical bit of information and is subject to improvement. The design of good heuristics is an empirical problem; judgment and intuition help, but the final measure of a heuristic must be its actual performance on problem instances.

If two states have the same or nearly the same heuristic evaluations, it is generally preferable to examine the state that is nearest to the root state of the graph. This state will have a greater probability of being on the *shortest* path to the goal. The distance from the

| | Tiles out of place | Sum of distances out of place | 2 x the number of direct tile reversals |
|---|---|---|---|
| 2 8 3<br>1 6 4<br>■ 7 5 | **5** | **6** | **0** |
| 2 8 3<br>1 ■ 4<br>7 6 5 | **3** | **4** | **0** |
| 2 8 3<br>1 6 4<br>7 5 ■ | **5** | **6** | **0** |

Goal:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | ■ | 4 |
| 7 | 6 | 5 |

Figure 4.14 Three heuristics applied to states in the 8-puzzle.

starting state to its descendants can be measured by maintaining a depth count for each state. This count is 0 for the beginning state and is incremented by 1 for each level of the search. This depth measure can be added to the heuristic evaluation of each state to bias search in favor of states found shallower in the graph.

This makes our evaluation function, f, the sum of two components:

$$f(n) = g(n) + h(n)$$

where g(n) measures the actual length of the path from any state n to the start state and h(n) is a heuristic estimate of the distance from state n to a goal.

In the 8-puzzle, for example, we can let h(n) be the number of tiles out of place. When this evaluation is applied to each of the child states in Figure 4.12, their f values are 6, 4, and 6, respectively, see Figure 4.15.

The full best-first search of the 8-puzzle graph, using f as defined above, appears in Figure 4.16. Each state is labeled with a letter and its heuristic weight, f(n) = g(n) + h(n). The number at the top of each state indicates the order in which it was taken off the open list. Some states (h, g, b, d, n, k, and i) are not so numbered, because they were still on open when the algorithm terminates.

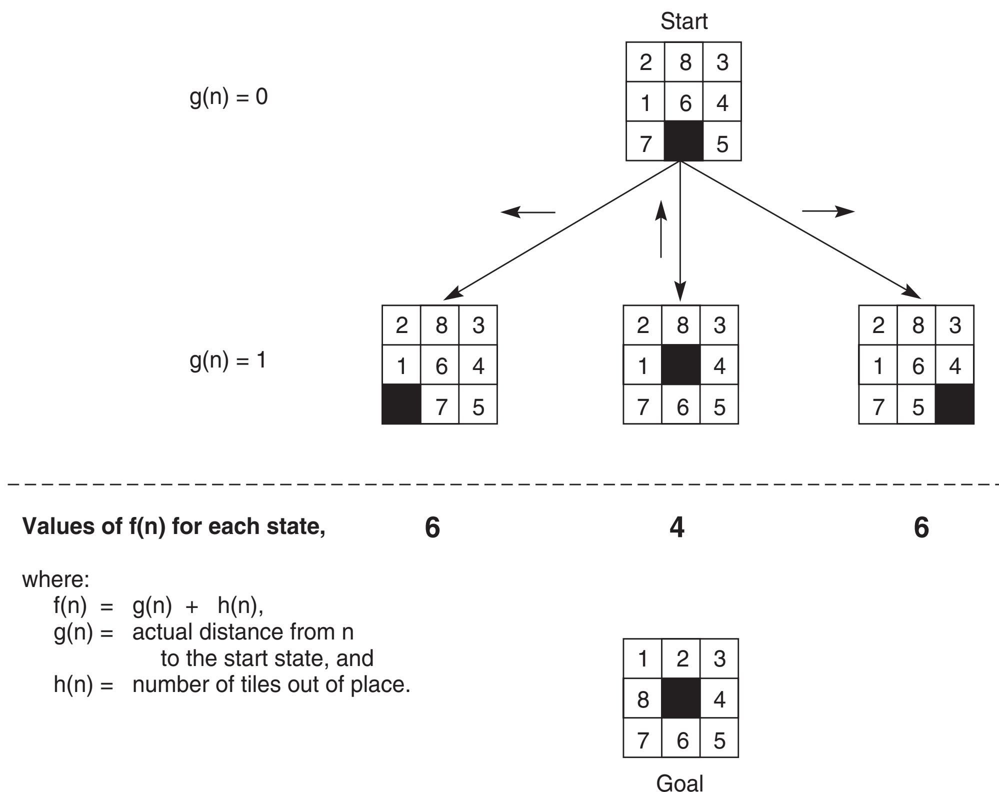



Figure 4.15 The heuristic f applied to states in the 8-puzzle.

The successive stages of open and closed that generate this graph are:

1. open = [a4];
   closed = [ ]
2. open = [c4, b6, d6];
   closed = [a4]
3. open = [e5, f5, b6, d6, g6];
   closed = [a4, c4]
4. open = [f5, h6, b6, d6, g6, i7];
   closed = [a4, c4, e5]
5. open = [ j5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5]
6. open = [l5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5, j5]
7. open = [m5, h6, b6, d6, g6, n7, k7, i7];
   closed = [a4, c4, e5, f5, j5, l5]
8. success, m = goal!

In step 3, both e and f have a heuristic of 5. State e is examined first, producing children, h and i. Although h, the child of e, has the same number of tiles out of place as f, it is one level deeper in the space. The depth measure, g(n), causes the algorithm to select f for evaluation in step 4. The algorithm goes back to the shallower state and continues to the goal. The state space graph at this stage of the search, with open and closed highlighted, appears in Figure 4.17. Notice the opportunistic nature of best-first search.

In effect, the g(n) component of the evaluation function gives the search more of a breadth-first flavor. This prevents it from being misled by an erroneous evaluation: if a heuristic continuously returns "good" evaluations for states along a path that fails to reach a goal, the g value will grow to dominate h and force search back to a shorter solution path. This guarantees that the algorithm will not become permanently lost, descending an infinite branch. Section 4.3 examines the conditions under which best-first search using this evaluation function can actually be guaranteed to produce the shortest path to a goal.

To summarize:

1. Operations on states generate children of the state currently under examination.
2. Each new state is checked to see whether it has occurred before (is on either open or closed), thereby preventing loops.
3. Each state n is given an f value equal to the sum of its depth in the search space g(n) and a heuristic estimate of its distance to a goal h(n). The h value guides search toward heuristically promising states while the g value can prevent search from persisting indefinitely on a fruitless path.

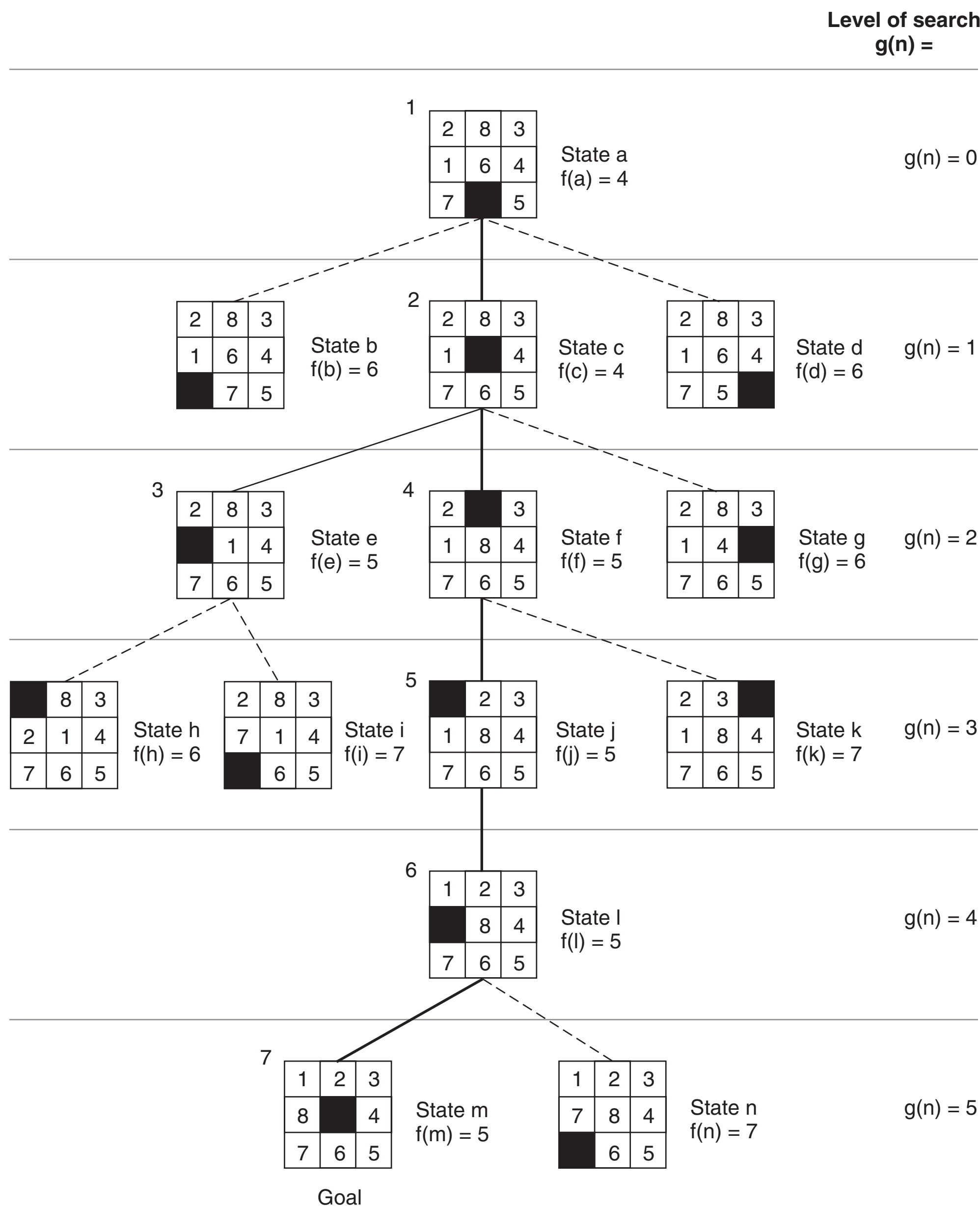



Figure 4.16 State space generated in heuristic search of the 8-puzzle graph.

4. States on open are sorted by their f values. By keeping all states on open until they are examined or a goal is found, the algorithm recovers from dead ends.

5. As an implementation point, the algorithm's efficiency can be improved through maintenance of the open and closed lists, perhaps as *heaps* or *leftist trees.*

Best-first search is a general algorithm for heuristically searching any state space graph (as were the breadth- and depth-first algorithms presented earlier). It is equally

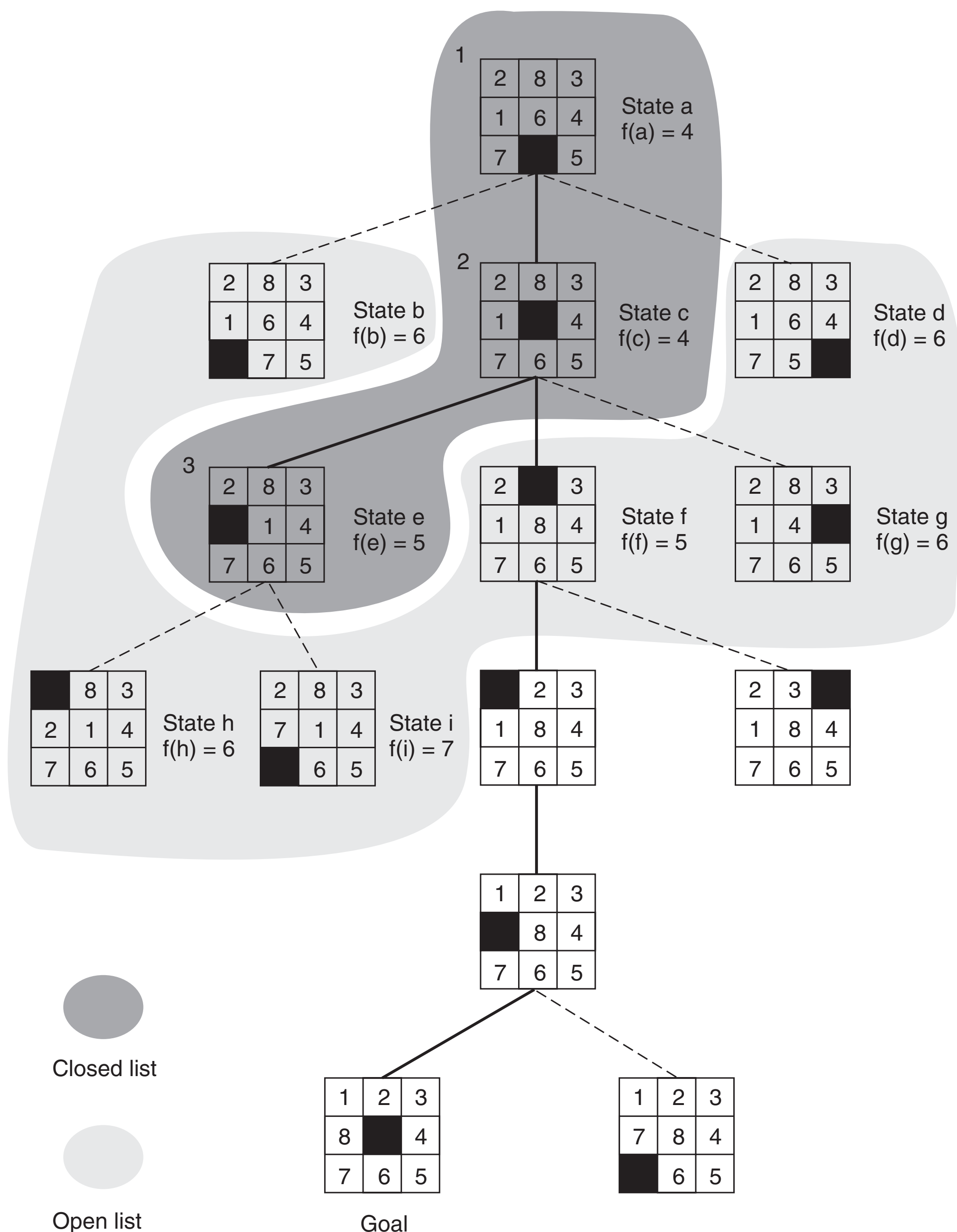



Figure 4.17 open and closed as they appear after the third iteration of heuristic search.

applicable to data- and goal-driven searches and supports a variety of heuristic evaluation functions. It will continue (Section 4.3) to provide a basis for examining the behavior of heuristic search. Because of its generality, best-first search can be used with a variety of heuristics, ranging from subjective estimates of state's "goodness" to sophisticated

measures based on the probability of a state leading to a goal. Bayesian statistical measures (Chapters 5 and 9) offer an important example of this approach.

Another interesting approach to implementing heuristics is the use of confidence measures by expert systems to weigh the results of a rule. When human experts employ a heuristic, they are usually able to give some estimate of their confidence in its conclusions. Expert systems employ *confidence measures* to select the conclusions with the highest likelihood of success. States with extremely low confidences can be eliminated entirely. This approach to heuristic search is examined in the next section.

### 4.2.3 Heuristic Search and Expert Systems

Simple games such as the 8-puzzle are ideal vehicles for exploring the design and behavior of heuristic search algorithms for a number of reasons:

1. The search spaces are large enough to require heuristic pruning.
2. Most games are complex enough to suggest a rich variety of heuristic evaluations for comparison and analysis.
3. Games generally do not involve complex representational issues. A single node of the state space is just a board description and usually can be captured in a straightforward fashion. This allows researchers to focus on the behavior of the heuristic rather than the problems of knowledge representation.
4. Because each node of the state space has a common representation (e.g., a board description), a single heuristic may be applied throughout the search space. This contrasts with systems such as the financial advisor, where each node represents a different subgoal with its own distinct description.

More realistic problems greatly complicate the implementation and analysis of heuristic search by requiring multiple heuristics to deal with different situations in the problem space. However, the insights gained from simple games generalize to problems such as those found in expert systems applications, planning, intelligent control, and machine learning. Unlike the 8-puzzle, a single heuristic may not apply to each state in these domains. Instead, situation specific problem-solving heuristics are encoded in the syntax and content of individual problem solving operators. Each solution step incorporates its own heuristic that determines when it should be applied; the pattern matcher matches the appropriate operation (heuristic) with the relevant state in the space.

**EXAMPLE 4.2.1: THE FINANCIAL ADVISOR, REVISITED**

The use of heuristic measures to guide search is a general approach in AI. Consider again the financial advisor problem of Chapters 2 and 3, where the knowledge base was treated as a set of logical implications, whose conclusions are either true or false. In actuality, these rules are highly heuristic in nature. For example, one rule states that an individual with adequate savings and income should invest in stocks: