The Elements of Artificial Intelligence

with Python

Steven L. Tanimoto
University of Washington

Draft version based upon The Elements of Artificial Intelligence Using Common Lisp

Part III: Chapter 5

February 1, 2005

Chapter 5

Search

5.1 The Notion of Search

Methods for searching are to be found at the core of many AI systems. Before knowledge representation became the key issue of AI in the 1970s, search techniques were at the center of attention in research and in courses on AI. They are still of central importance in AI not only because most systems are built around them but because it is largely through an understanding of search algorithms that we are able to predict what kinds of AI problems are solvable in practice.

The idea of searching for something implies moving around examining things and making decisions about whether the sought object has yet been found. A cave explorer searching for treasure moves around from one underground place to another, looking to see where he is going and whether there are any valuables around him. He is constrained by the geometry (or topology) of the cave; he must follow passageways provided by nature. A chess player searching for the best move in the middle of a game mentally makes moves and countermoves, finds the merits of resulting board positions, and in the process determines the degree to which she can control the game to reach some of these positions. An engineer who designs integrated-circuit layouts considers sequences of design choices that lead to acceptable arrangements of components and electrical connections; he searches through spaces of possible designs to find those that have the required properties.

Many computer programs must also search along constrained paths through intricate networks of knowledge, states, or conditions to find important information or to reach a goal position. In general such a network can be described as a graph: a set of nodes together with a set of arcs that connect pairs of nodes. The nodes represent "states" of a space of possible configurations. The transitions or moves that go from one state to another are represented by arcs of the graph.

The choice of a search method is often a critical choice in the design of an

AI program. A poor choice can ensure that the program will always flounder in the "combinatorial quagmire" before it can find a solution to any nontrivial problem. Exhaustive techniques can be appropriate for small problems, but the human user will become exhausted waiting for a machine that uses a "British Museum" search method on only a moderately complicated problem. A good search method typically uses some particular information about the problem or some general knowledge to focus the search for a solution on areas of the state space that have a reasonable chance of containing the solution.

In AI systems, the networks to be searched may be represented either explicitly or implicitly. For example, the inclusion hierarchy used by the Linneus program in Chapter 4 is an explicit network, where each arc is stored using relational information on the property list of an atom. However, many AI programs must search networks whose arcs are not explicitly stored and which must be generated from rules, one at a time. A computer program that plays chess explores a game tree in which each node corresponds to a board position and each arc corresponds to a legal move. The complete tree of all possible games is so huge that it cannot be explicitly represented. Rather, parts of it must be generated when needed according to the rules for moving pieces. The goal, of course, is to choose the best moves by evaluating the consequences of each possible move from the current position. This search is for more than just a good node in the tree; it is, in a sense, for the best "subtree" from the current position. In this chapter we work with examples of both implicitly and explicitly represented search spaces.

There are several key notions connected with search. The most important of them, just mentioned above, is the concept of a *state space*: the set of all the possible states for a problem together with the relations involving states implied by the moves or operators. The graph whose nodes represent states and whose arcs represent the relations provides a good abstract representation for a state space. Another key notion is that of a *move generator*: a way to obtain the successors of a given state or node. A third key notion is the *search method*, or kind of algorithm that is used to control the exploration of the state space. A fourth idea is that of *search heuristics*: these are guiding principles, often of a pragmatic nature, that tend to make the search easier. A common way of controlling a search is to employ an "evaluation function," which computes an estimate of the merits of a particular successor to the current state or node.

Puzzles are easy to understand and describe, and they provide a good starting point for studying search methods. We shall begin our discussion of search with an analysis of a simple class of puzzles called "painted squares" puzzles. Although this class of puzzles is elementary, it is a good vehicle for examining some subtle issues in the description of operators. The class is general, in that particular versions of the puzzle may have zero, one, or many solutions. This feature makes the puzzles somewhat more realistic as problems than the Fifteen or Towers of Hanoi puzzles, which are sometimes used for teaching elementary search techniques.

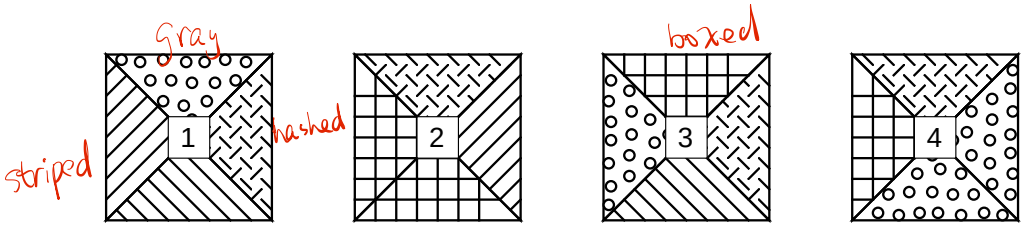

Figure 5.1: Pieces for one of the painted squares puzzles.

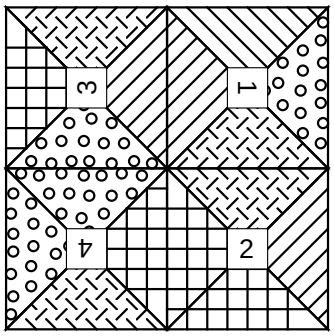

Figure 5.2: One solution to the sample puzzle.

We will describe a procedure for solving painted squares puzzles that works by searching a space of configurations (which may be regarded as potential solutions).

5.2 Painted Squares Puzzles

5.2.1 A Kind of Geometrical Puzzle

Given a set of N square blocks with painted sides, it is desired to place the squares adjacent to one another to form one large rectangle (of given dimensions), so that adjacent sides of the squares always match. A sample game for $N = 4$ and dimensions 2 by 2 is shown in Figures 5.1 and 5.2.

We should note that such puzzles may or may not have solutions, depending on how the squares have been painted. For example, if all four sides of piece 1 were striped, all four sides of 2 boxed, all four sides of 3 gray, and all four sides of 4 hashed, no two squares could be placed together, let alone all four of them.

5.2.2 Solution Procedure

A solution, when one exists, can be found by the following procedure: enumerate the vacant positions in the rectangle with the numbers 1 to N, using a left-to-right, top-to-bottom order. Starting with vacant position 1, select the first painted square and place it in its first orientation (each square can be rotated into four different orientations). At each successive step, after having filled the ith vacancy with the jth square in the kth orientation, attempt to fill the $i+1$st vacancy with the lowest-numbered unused square in the first orientation. If this new placement does not match sides with previously placed squares, the following alterations are tried: successive orientations of the same square, successive unused squares, and retraction of the most recently placed square (backtracking). This procedure eventually either finds a solution if one exists, or exhausts all the possibilities without finding a solution. Backtracking can often be very inefficient. In the case of the painted squares puzzles it is generally much more efficient than an obvious alternative, which is to generate all possible arrangements of the squares in the rectangular space and test each one to see whether it is a solution to the puzzle!

5.2.3 States and Operators

We may think of each partial (or complete) arrangement of squares in vacancies as a "state." A *state* is one of the possible configurations of the puzzle. It is a snapshot of one situation. A state is a configuration that could be reached using a sequence of legal moves or decisions, starting from an initial configuration. The sequence of steps that leads to a state is not part of the state. There may be more than one way to get to the same state.

Once again, the *state space* for a puzzle consists of the set of all the states for a puzzle together with the relation implied by the legal moves for the puzzle. Each legal move from a state leads to a single new state. The pair (current state, new state) form one element in the move relation. For the painted squares puzzle of Figure 5.1, a portion of the state space is shown in Figure 5.3.

The *representation* of a state is an embodiment of the essential information about a state; the representation distinguishes the state from all others. A good representation also does more: it facilitates the application of operators to compute successive states, it is understandable by the scientist or programmer, it is efficient, etc. In the painted squares puzzle, we may represent a state by a structure that represents the board and the pieces on the board. The board could be an array, each of whose elements is either a code for "vacancy" or a pair: [piece, orientation]. For example, piece 1 in orientation 2 (i.e., rotated 180 degrees counterclockwise) is represented by the pair [⟨Piece 1⟩, 2].

A state can be changed by placing a new square into any vacant square on the board or, perhaps, by removing the most recently placed square. More generally, we say that states are changed by applying operators. Operators are

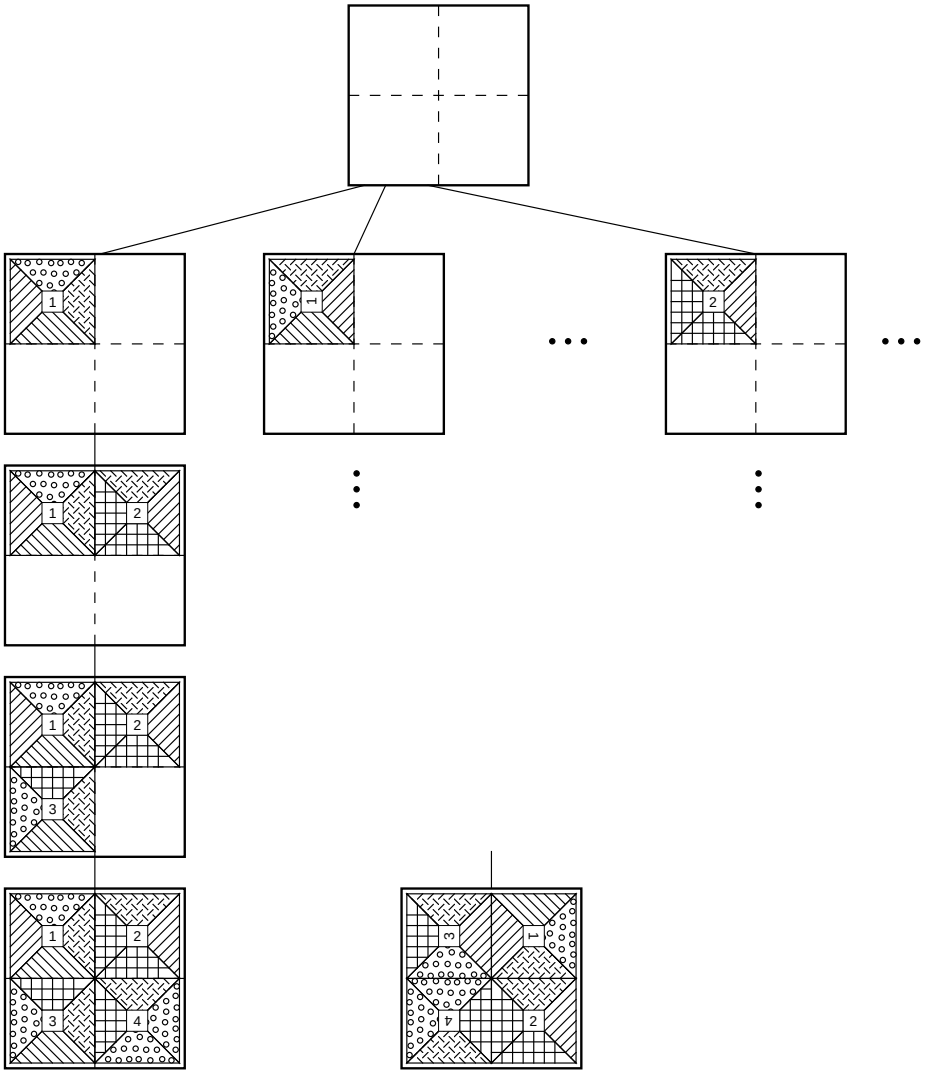

Figure 5.3: Portion of the state space for the sample puzzle.

transformations that map one state to another.

An example operator is "place piece 3 in the second place with the third orientation." This operator is a rather specific one and is applicable only if the current state satisfies some stringent conditions: the first place must be full; the second place must be empty; piece 3 must not have been used already to fill place 1; and when rotated to the third orientation, the side of piece 3 that would touch the piece in place 1 must match there.

A more general operator is "place the first unused piece in the next vacant position in the first orientation that makes all the adjacent pieces match." There are still conditions for the applicability of this operator, but they are not so restricting. There must be an unused piece (if not, the puzzle has been solved!), and there must be at least one orientation of the piece such that its placement in the next vacant position does make all adjacent sides match.

There is obviously quite a variety of operators one could imagine that could be used to describe moves in our puzzle. To put some order on them, we will consider general schemes for making state changes that may "generate" operators when called upon. The backtracking procedure sketched out earlier does essentially that. It produces new operators in an orderly sequence in order to search for a goal state.

5.2.4 Painted Squares in Python

Here is a program that illustrates a possible representation for pieces and states, along with a recursive, depth-first search method to find solutions.

We start with a class to represent the pieces of the puzzle.

```
class piece:
    # constants:
    striped = 1
    hashed = 2
    gray = 3
    boxed = 4
    textureNames = ['striped','hashed','gray','boxed']

    #constructor:
    def __init__(self, n, p):
        #instance variables:
        self.number = n
        self.pattern = p

    def number(self): return self.n

    def pattern(self): return self.pattern
```

```
    def describe(self):
        return "Piece " + str(self.number) + " (" +\
            str(map(lambda x: piece.textureNames[x-1], self.pattern)) + ")"

# Here are the particular pieces used in the example puzzle.
pieces = [
    piece(1, [piece.gray,   piece.striped, piece.striped, piece.hashed]),
    piece(2, [piece.hashed, piece.boxed,   piece.boxed,   piece.striped]),
    piece(3, [piece.boxed,  piece.gray,    piece.striped, piece.hashed]),
    piece(4, [piece.hashed, piece.boxed,   piece.gray,    piece.gray]) ]

'''piece 1 looks like this:
+-------------------------+
|          gray           |
|striped     1     hashed |
|          striped        |
+-------------------------+        '''
```

Next is another class. This one represents states for the state-space search.

```
class state:
    boardWidth = 2
    boardLength = 2
    boardArea = boardWidth * boardLength

    def __init__(self, b, up, vl):
        self.board = b
        self.unusedPieces = up
        self.vacanciesLeft = vl

    def describe(self):
        pieceNumbers = map(lambda p: p.number, self.unusedPieces)
        return "State with board; " + showBoard(self.board) +\
               "; unused pieces: " + str(pieceNumbers) +\
               "; vacancies: " + str(self.vacanciesLeft)
```

The next piece of code is for printing out states.

```
rowOffsets = [0,1,2,1]
colOffsets = [1,0,1,2]
textureAbbreviation = ['s', 'h', 'g', 'b']

def drawOrientedPiece(thePiece, theOrientn, theArray, row, col):
    opat = orient(thePiece, theOrientn)
    global rowOffsets
```

```
    global colOffsets
    for k in range(4):
        theArray[4*row + 1 + rowOffsets[k]]\
                [4*col + 1 + colOffsets[k]] =\
                       textureAbbreviation[opat[k]-1]
    theArray[4*row+2][4*col+2] = str(thePiece.number)

def showBoard(b):
    line = ['+','-','-','-','+','-','-','-','+']
    charRow = ['|',' ',' ',' ','|',' ',' ',' ','|']
    charArray = [line, charRow, charRow[:], charRow[:],
                 line, charRow[:], charRow[:], charRow[:], line]
    for i in range(state.boardLength):
        for j in range(state.boardWidth):
            if not b[i][j] == 0:
                drawOrientedPiece(b[i][j][0], b[i][j][1],
                                  charArray, i, j)
    for r in charArray:
        line = reduce(lambda c,d: c+d, r)
        print line
    return ''
```

Next are the functions that implement the search.

```
def createInitialState():
    board = [[0, 0], [0, 0]] # all vacant
    global pieces
    return state(board, pieces, state.boardArea)

def placePiece(currentState, piece, row, column, orientation):
    newState = state(currentState.board[:],
                     removePiece(piece, currentState.unusedPieces),
                     currentState.vacanciesLeft - 1)
    newState.board[row][column] = [piece, orientation]
    return newState

def removePiece(p, lst):
    lstCopy = lst[:]
    lstCopy.remove(p)
    return lstCopy

def orient(piece, orientation):
    return rotateList(piece.pattern, orientation)

def rotateList(lst, n):
```

```
    k = len(lst)
    return map(lambda i: lst[(i-n) % k], range(k))

def sidesOK(newPiece, orientation, row, col, currentState):
    trialOrientedPattern = orient(newPiece, orientation)
    whichNeighbor = 0  # 0 is north, 1 is west, etc.
    okSoFar = True
    for displacement in [[-1, 0],[0, -1],[1, 0],[0, 1]]:
        neighborRow = row + displacement[0]
        neighborCol = col + displacement[1]
        thisSideOK = \
          (neighborRow < 0) or\
          (neighborRow >= state.boardLength) or\
          (neighborCol < 0) or\
          (neighborCol >= state.boardWidth) or\
          matchSides(\
              trialOrientedPattern,
              whichNeighbor,
              orientedPieceAt(neighborRow,\
                              neighborCol,\
                              currentState),\
              oppositeDir(whichNeighbor))
        okSoFar = okSoFar and thisSideOK
        whichNeighbor += 1
    return okSoFar

def oppositeDir(direction):
    return (direction + 2) % 4

def orientedPieceAt(row, col, theState):
    return theState.board[row][col]

def matchSides(orientedPat1, whichSide1,
               orientedPiece2, whichSide2):
    if (orientedPiece2 == 0): return True
    val = orientedPat1[whichSide1] ==\
           orient(orientedPiece2[0],\
                  orientedPiece2[1])[whichSide2]
    return val

def solveSquares(currentState):
    if currentState.vacanciesLeft == 0:
        show(currentState)
        return True
```

```
    else:
        k = state.boardArea - currentState.vacanciesLeft
        nextRow = k / state.boardWidth
        nextCol = k % state.boardWidth
        for p in currentState.unusedPieces:
            tryPiece(p, nextRow, nextCol, currentState)

count = 0 # number of solutions found so far

def show(soln):
    global count
    count += 1
    print 'Solution ' + str(count) + ':'
    showBoard(soln.board)

def tryPiece(p, row, col, currentState):
    for orientation in range(4):
        tryOrientation(orientation, p, row, col, currentState)

def tryOrientation(orientn, p, row, col, currentState):
    if sidesOK(p, orientn, row, col, currentState):
        newState = placePiece(currentState, p, row, col, orientn)
        return solveSquares(newState)
    else: return False
```

Here is code to start the puzzle solver.

```
def test():
    s0 = createInitialState()
    solveSquares(s0)

test()
```

Here is what gets printed out by the program.

```
Solution 1:
+---+---+
| g | s |
|s1h|h2b|
| s | b |
+---+---+
| s | b |
|h3g|g4h|
| b | g |
+---+---+
```

```
Solution 2:
+---+---+
| b | s |
|b2h|h3g|
| s | b |
+---+---+
| s | b |
|s1g|g4h|
| h | g |
+---+---+
Solution 3:
+---+---+
| b | g |
|b2h|h4g|
| s | b |
+---+---+
| s | b |
|s1g|g3h|
| h | s |
+---+---+
Solution 4:
+---+---+
| b | s |
|g3h|h2b|
| s | b |
+---+---+
| s | b |
|s1g|g4h|
| h | g |
+---+---+
```

5.3 Graph Search Techniques

5.3.1 A Search Problem

In order to illustrate several different search techniques, in the general context of graphs, we shall use a map of cities of France, transformed into a graph. This graph serves as an *explicit* representation of a state space. The search techniques we describe will be general, and can also work in a state space that is implicit. In that case, the states would be generated gradually by applying operators rather than pre-existing. (When search techniques are used for theorem proving in Chapter 6, the search space is implicitly represented.)

Here, the use of an explicit state space helps us to understand the behavior of

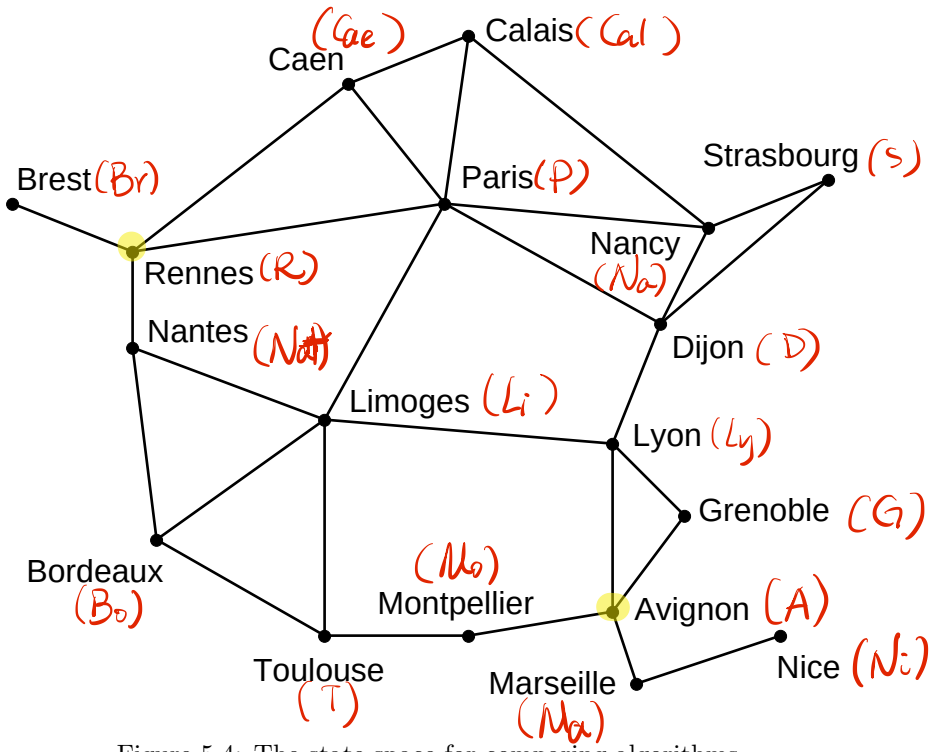

Figure 5.4: The state space for comparing algorithms.

the search algorithms by making it obvious (to us humans) what kinds of paths to a solution exist. This is particularly helpful when alternative algorithms are compared, as is the case in this chapter. In our space, the roles of states are played by cities (the nodes of the graph, and the nodes, in turn, can be represented as strings in lists), and the moves are transitions along arcs of the graph. Although we put an explicit representation of the graph into the machine, our search procedures only have access to a portion of the representation at any particular time. The algorithms search from a starting point in the graph outward to find a goal node. The algorithms may or may not actually explore the entire graph; for example, if the goal is found after only a few transitions, the majority of the nodes may remain unexplored. Figure 5.4 illustrates the search space.

We may represent the adjacency data in the map by listing for each city the other cities directly connected to it and putting this information into a hash table. Let's put the following into a file named `france.py` so that we can easily import it later in various search programs.

```
ADJ = {}
```

```
ADJ['Brest'] = ['Rennes']
ADJ['Rennes'] = ['Caen','Paris','Brest','Nantes']
ADJ['Caen'] = ['Calais','Paris','Rennes']
ADJ['Calais'] = ['Nancy','Paris','Caen']
ADJ['Nancy'] = ['Strasbourg','Dijon','Paris','Calais']
ADJ['Strasbourg'] = ['Dijon','Nancy']
ADJ['Dijon'] = ['Strasbourg','Lyon','Paris','Nancy']
ADJ['Lyon'] = ['Grenoble','Avignon','Limoges','Dijon']
ADJ['Grenoble'] = ['Avignon','Lyon']
ADJ['Avignon'] = ['Grenoble','Marseille','Montpellier','Lyon']
ADJ['Marseille'] = ['Nice','Avignon']
ADJ['Nice'] = ['Marseille']
ADJ['Montpellier'] = ['Avignon','Toulouse']
ADJ['Toulouse'] = ['Montpellier','Bordeaux','Limoges']
ADJ['Bordeaux'] = ['Limoges','Toulouse','Nantes']
ADJ['Limoges'] = ['Lyon','Toulouse','Bordeaux','Nantes','Paris']
ADJ['Nantes'] = ['Limoges','Bordeaux','Rennes']
ADJ['Paris'] = ['Calais','Nancy','Dijon','Limoges','Rennes','Caen']
```

Let us suppose that we wish to find, with the help of a program, a route from Rennes to Avignon. There are several approaches we might take.

5.3.2 Hypothesize and Test

If one has very little knowledge about the space one must search, one may be inclined to explore it at random. Beginning at Rennes, one could roll dice to choose which road to begin with. It is possible that a random decision sequence might lead one to the goal state (Avignon). Without an informed way of generating hypotheses (trial routes), however, one can expect to waste a lot of time with this approach. The first obvious improvement we can make is to systematize the search so that the various possible alternatives are tried in an orderly fashion.

5.3.3 Depth-First Search

We can proceed by listing the cities directly accessible from Rennes and then going to the first of these (taking us to Caen), adding to the list the new places one could go to directly from there, and moving to the first of those, and so on, until we have either found Avignon or reached a dead end. If a dead end is reached, we can back up to the last city visited from which we had a choice of new cities and try the next alternative.

The depth-first algorithm that we employ here is almost the same as the iterative one given for the painted squares puzzle. The one essential difference now is that we are interested in finding the *path* to the goal node or state rather

than a state description of the goal node or state. The version here finds only the first path that might exist, and it does not continue searching after this path is found. We also have altered the terminology slightly to speak in terms of "nodes" rather than "states." We can describe this *depth-first* search algorithm as follows:

1. Put the start node on a list OPEN and associate a null pointer with the node.

2. If OPEN is empty, output "FAILURE" and stop.

3. Select the first node on OPEN and call it N. Delete it from OPEN and put it on a list CLOSED. If N is a goal node, output the list obtained by following the chain of pointers beginning with the pointer associated with N.

4. Generate the list L of successors of N and delete from L those nodes already appearing on the list CLOSED.

5. Delete any members of OPEN that occur on L. Concatenate L onto the front of OPEN and to each node in L associate a pointer to N.

6. Go to step 2.

This may be described in Python as follows:

```
from france import ADJ

PREDECESSOR = {}  # initialize a hash of backward links
COUNT = 0

def dfgs(start, goal):
    OPEN = [start]                    # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    while OPEN != []:                 # Step 2
        n = OPEN[0]                   # Step 3
        OPEN = OPEN[1:]
        CLOSED.append(n)
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)
        lst = successors(n)           # Step 4
        lst = listDifference(lst, CLOSED)
        # successors are added to the beginning of OPEN.
        # If any successor was already on OPEN, move it up.
        OPEN = lst +\
            listDifference(OPEN, lst) # Step 5
        for elt in lst:
            PREDECESSOR[elt] = n
    # end of loop. This is implicitly # Step 6
```

The helping function `extractPath` follows the pointers to produce a list of nodes on the path found:

```
def extractPath(n):
    if PREDECESSOR[n]==None: return [n]
    return extractPath(PREDECESSOR[n]) + [n]
```

The function `successors` gets the cities adjacent to `city`:

```
def successors(city):
    return ADJ[city]
```

The function `listDifference` is like a set-difference operation that preserves the ordering among the remaining elements.

```
def listDifference(list1, list2):
    'Return a copy of list2 with all elements in list2 removed.'
    list3 = list1[:]
    for elt in list2:
```

```
        if elt in list3: list3.remove(elt)
    return list3
```

The function `test` can be used to test the program.

```
def test():
    start = 'Rennes'
    goal = 'Avignon'
    print "Starting a depth-first graph search from " + start
    path = dfgs(start, goal)
    print path
    global COUNT
    print str(COUNT) + " nodes expanded during the search."
```

One problem with this method of finding a path is that it doesn't necessarily find the shortest one. Let us make the simple assumption (for the moment) that the length of a route is equal to the number of arcs in the graph along the route. The depth-first search method may just as well find a longest path between two points as a shortest path. In fact, running the `test` function gives us the rather roundabout route

```
['Rennes', 'Caen', 'Calais', 'Nancy', 'Strasbourg', 'Dijon',
  'Lyon', 'Grenoble', 'Avignon']
```

which with eight arcs (and nine cities) is four arcs longer than necessary.

The computation time required by depth-first search depends on several factors. In this formulation, these factors include the number of nodes visited (i.e., the number of iterations in the main loop) and the time required by the function `listDifference`. The first of these is generally considered most important.[1]

We can alter our searching procedure to find shortest paths; we shall turn the function into a "breadth-first" search procedure.

5.3.4 Breadth-First Search

A more conservative style of searching a graph is to search along all paths of length 1 from the start node, then along all paths of length 2, length 3, etc., until

[1] In this implementation, number of nodes and the efficiency of `listDifference` are key. The product of these can be as bad as $O(n^3)$ where n is the number of nodes in the graph. With a more efficient organization, an $O(e)$ algorithm is possible where e is the number of edges (arcs) in the graph; this is $n(n-1)$ or $O(n^2)$ in the worst case. However, that requires departing from the simple algorithm structure that has become standard in AI for the description of ordered search algorithms including depth-first search, breadth-first search, best-first search, uniform-cost search, and A* search. The $O(e)$ algorithm also makes the assumption that it is possible to test in constant time whether a given state has been visited before; this is not usually a realistic assumption in AI, because we do not begin with an enumerated set of states (to which we can associate a "mark" array) but must generate states as the search progresses. For an $O(e)$ formulation, see [Aho et al. 1983], pp. 215-217.

either the goal is found or the longest possible acyclic paths have been tried. In actuality, when we search along paths of length k, we need not reexamine the first $k-1$ nodes of each such path; we need only take one step farther in each possible direction from the nodes newly reached in iteration number $k-1$. This method is called *breadth-first* search.

A minor modification in our function `dfgs` makes it become `breadthfgs`. In steps 4 and 5, we change the way the list L is merged with the list OPEN. Rather than concatenate L at the front of OPEN, we put it at the back. That is, we effectively replace

```
OPEN = lst + listDifference(OPEN, lst)
```

with

```
OPEN = OPEN + listDifference(lst, OPEN)
```

and thus put the newly reached nodes on the *end* of OPEN, for processing after all the nodes at the current depth are finished. The resulting function definition is

```
def breadthfgs(start, goal):
    OPEN = [start]
    CLOSED = []
    PREDECESSOR[start] = None
    while OPEN != []:
        n = OPEN[0]
        OPEN = OPEN[1:]
        CLOSED.append(n)
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)
        lst = successors(n)
        lst = listDifference(lst, CLOSED)
        lst = listDifference(lst, OPEN)
        # the new successors are added to the end of OPEN:
        OPEN = OPEN + lst
        for elt in lst:
            PREDECESSOR[elt] = n
```

Now, the possible routes from the start node are explored in order of increasing length, so that as soon as the goal node is found, we know we have constructed a minimum-length path to it. The result of evaluating

```
breadthfs('Rennes', 'Avignon')
```

is the much more reasonable route

```
['Rennes', 'Paris', 'Dijon', 'Lyon', 'Avignon']
```

The algorithm expands 15 nodes in the course of finding this path.

Of course it is possible that `dfgs` might stumble on the shortest path by some coincidence, and it might do so in many fewer iterations than `breadthfs` requires. On the other hand, if the path is short, `dfgs` might not find the goal right away, whereas `breadthfs` would find it very quickly.

5.4 Heuristic Search Methods

5.4.1 Evaluation Functions

Both the depth-first and breadth-first methods are "blind" in the sense that they use exhaustive approaches that can't "see" where they are going until they get there. That is, they don't have any sense of where the goal node lies until they find it. Consequently, they often spend a lot of time searching in totally fruitless directions. If some general guiding information is available, the searching can be biased to move in the general direction of the goal from the very beginning. For example, when a friend hides a present for you in the house and then gives you clues of "warmer" and "colder" as you move closer or farther away from the cache, you will have a much easier time locating the present than without such feedback.

A function f that maps each node to a real number and serves to estimate either the relative benefit or the relative cost of continuing the search from that node is an *evaluation function*. In the remainder of this section, we consider only evaluation functions that are cost functions. Typically f(N) is designed to estimate the distance remaining between N and the goal node. Alternatively f(N) might estimate the length of a path from the start node to the goal node that passes through N. The evaluation function is used to decide the order in which nodes are to be considered during the search. A search method that tends to first expand nodes estimated to be closer to the goal is likely to reach the goal with fewer steps.

For the problem of finding a route from Rennes to Avignon, we now introduce an evaluation function to provide some rough guidance to the search. Suppose that the longitude of each city is available. We make f(N) = `longitudediff(N, Avignon)`, which we define to be the absolute value of the difference between the longitude of N and the longitude of Avignon. We can now arrange to have the nodes on OPEN kept ordered by increasing f value, so that the most promising node to process next always appears at the front. The procedure for exploiting the new information, called "best-first" or "ordered" search, is described in the next section. Before proceeding there, we give the additional Python representations that will be required.

First we set up a hash table and store the longitude data required in computing f:

```
LONGITUDE = {'Avignon':48, 'Bordeaux':-6, 'Brest': -45, 'Caen':-4,
             'Calais': 18, 'Dijon':51, 'Grenoble':57, 'Limoges':12,
             'Lyon':48, 'Marseille':53, 'Montpellier':36,
             'Nancy':62, 'Nantes': -16, 'Nice':73, 'Paris':23,
             'Rennes': -17, 'Strasbourg': 77, 'Toulouse':14}
```

This information can be kept in a separate file `france2.py`.

We need one more hash table FVALUE to remember the heuristic value at each node visited. Let's initialize that hash table here:

```
FVALUE = {}
```

Now we define two functions used in computing f:

```
# longitudeDiff returns the absolute value of the
# difference in longitudes between nodes n1 and n2
# in tenths of a degree.
def longitudeDiff(n1, n2):
    return abs(LONGITUDE[n1] - LONGITUDE[n2])

# f evaluates the difference in longitude between
# the current node n and the goal node.
def f(n):
    global GOAL
    return longitudeDiff(n, GOAL)
```

We know that GOAL will be bound to 'Avignon' when the search for 'Avignon' from 'Rennes' is begun.

5.4.2 Best-First (Ordered) Search

Let us first describe the general procedure for best-first search and then discuss its application to increasing the efficiency of finding a route from Rennes to Avignon.

With the aid of an evaluation function f on the nodes of a graph, it is desired to find a goal node starting from a start node S.

We begin by placing node S on a list called OPEN. Then we successively process the node(s) on OPEN by examining them to see if they are goal nodes and, if not, transferring them to another list CLOSED while placing their successors on OPEN, all the time avoiding redundant processing, updating f(N) for each node N processed, and treating nodes on OPEN in an order that gives priority to the node having lowest f(N). More precisely:

1. Place the starting node S on OPEN, compute f(S), and associate this value with S. Associate a null pointer with S.

2. If OPEN is empty, return "FAILED" and stop or exit.

3. Choose a node N from OPEN such that $f(\mathrm{N}) \leq f(\mathrm{M})$ for each M on OPEN and such that N is a goal node if any goal node achieves that minimum f value.

4. Put N on CLOSED and remove N from OPEN.

5. If N is a goal node, return the path from S to N obtained by tracing backward the pointers from N to S. Then stop or exit.

6. For each successor J that is already on OPEN, recompute $f(\mathrm{J})$ and compare it with its previous f value. If the new value is smaller, associate this new value with J, reposition J on OPEN, and redirect the pointer from J to point back to N.

7. For each successor J of N that is not already on OPEN or on CLOSED:

   (a) Compute $f(\mathrm{J})$ and associate it with J.

   (b) Put J on OPEN.

   (c) Associate a pointer with J pointing back to N.

8. Go to Step 2.

This general procedure may be used to search arbitrary graphs or trees such as those that describe the possible sequences of moves in puzzles. The search is said to be "ordered" (or "best-first") because at each iteration of Step 3, it chooses the best or one of the best alternative directions for searching, according to the evaluation function f. The efficiency of the search depends on the quality of this function. This function should yield relatively low values along the shortest path to a goal node if it is to make the search efficient.

Let us now apply the best-first searching method to find Avignon from Rennes, using `longitudeDiff` as the basis for evaluating nodes on OPEN. Here is the main procedure. It is preceded by initializations for a couple of global variables.

```
PREDECESSOR = {}
COUNT = 0

def bestfs(start, goal):
    "Performs a best-first search from start for goal."
    OPEN = [start]                    # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    while OPEN != []:                 # Step 2
        n = deleteMin(OPEN, f)        # Step 3
        OPEN = OPEN[1:]
        CLOSED.append(n)
```

```
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)
        lst = successors(n)           # Step 4
        lst = listDifference(lst, CLOSED)
        # successors are added to OPEN.
        OPEN = lst +\
            listDifference(OPEN, lst) # Step 5
        for elt in lst:
            PREDECESSOR[elt] = n
    # end of loop. This is implicitly # Step 6
```

The main procedure employs an important helping function. This function is `deleteMin`. The function `deleteMin` finds the "best" element of the given list according to the heuristic evaluation function. It deletes it from the list and returns it.

```
def deleteMin(lst, fn):
    minVal = 9999  # a very large value
    minElt = None
    for e in lst:
        temp = fn(e)
        if temp < minVal: minVal = temp; minElt = e
    lst.remove(minElt)
    return minElt
```

In order to test `bestfs`, we define a new function `test`.

```
def test():
    start = 'Rennes'
    goal = 'Avignon'
    print "Starting a depth-first graph search from " + start
    path = bestfs(start, goal)
    print path
    global COUNT
    print str(COUNT) + " nodes expanded during the search."
```

As with `breadthfs`, we get the following path, which consists of four arcs and is a shortest path:

```
['Rennes', 'Paris', 'Dijon', 'Lyon', 'Avignon']
```

The number of nodes expanded during this test is 13. Thus, the incorporation of the evaluation function led to an improvement over the number, 15, of nodes expanded by `breadthfs` (a blind method).

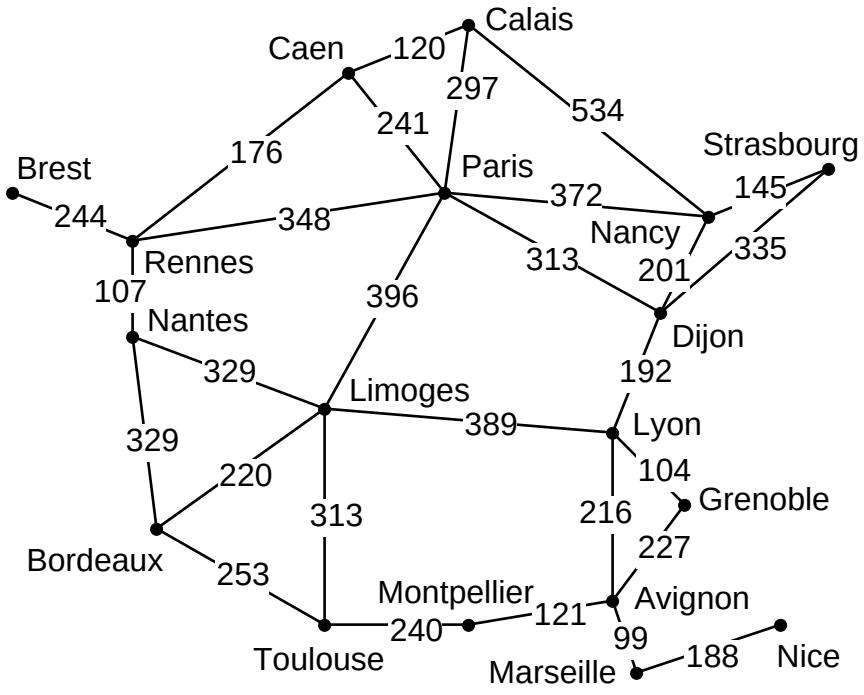

Figure 5.5: State space graph with distance ("cost") information.

5.4.3 Searching Graphs with Real Distances

In the previous section, we made the assumption that the length of a route from one city to another is equal to the number of arcs in the path. The actual road distances were not considered. These distances would, of course, provide a much better basis for the problem of finding a shortest driving route than the number of graph arcs on the path do. Assuming that we have a measure of the kilometers along each arc of the graph, the total distance for a route is the sum of the distances of the arcs that constitute it.

Adding distance information to our graph of French cities, we have the labeled graph of Figure 5.5.

We may represent this labeled graph by using a hash table DISTANCE of hash tables. There is a separate hash table for each city's distances to the other cities. We'll put this information in a file `france3.py`.

```
# Here are actual intercity distances from the Michelin map:
DISTANCE = {}
DISTANCE['Brest'] = {'Rennes':244}
```

```
DISTANCE['Rennes'] = {'Caen':176,'Paris':348,'Brest':244,'Nantes':107}
DISTANCE['Caen'] = {'Calais':120,'Paris':241,'Rennes':176}
DISTANCE['Calais'] = {'Nancy':534,'Paris':297,'Caen':120}
DISTANCE['Nancy'] = {'Strasbourg':145,'Dijon':201,'Paris':372,'Calais':534}
DISTANCE['Strasbourg'] = {'Dijon':335,'Nancy':145}
DISTANCE['Dijon'] = {'Strasbourg':335,'Lyon':192,'Paris':313,'Nancy':201}
DISTANCE['Lyon'] = {'Grenoble':104,'Avignon':216,'Limoges':389,'Dijon':192}
DISTANCE['Grenoble'] = {'Avignon':227,'Lyon':104}
DISTANCE['Avignon'] = {'Grenoble':227,'Marseille':99,'Montpellier':212,\
 'Lyon':216}
DISTANCE['Marseille'] = {'Nice':188,'Avignon':99}
DISTANCE['Nice'] = {'Marseille':188}
DISTANCE['Montpellier'] = {'Avignon':212,'Toulouse':240}
DISTANCE['Toulouse'] = {'Montpellier':240,'Bordeaux':253,'Limoges':313}
DISTANCE['Bordeaux'] = {'Limoges':220,'Toulouse':253,'Nantes':329}
DISTANCE['Limoges'] = {'Lyon':389,'Toulouse':313,'Bordeaux':220,\
 'Nantes':329,'Paris':396}
DISTANCE['Nantes'] = {'Limoges':329,'Bordeaux':329,'Rennes':107}
DISTANCE['Paris'] = {'Calais':297,'Nancy':372,'Dijon':313,\
 'Limoges':396,'Rennes':348,'Caen':241}
```

In order to apply the breadth-first search method here, it should be modified so that it opens new nodes in order of their minimum distance from the start node. Thus, each time a successor node M of a node N is generated, we should (1) see whether M is on CLOSED and, if so, not consider it further; but if not, (2) we should compute its distance from the start node along the path just followed as Temp = NodeDistance(N) + ArcDistance(M, N); (3) examine OPEN for an occurrence of M and, if present, we should compare the value of NodeDistance(M) with Temp and, if Temp is smaller, delete the old occurrence of M on OPEN; and (4) set NodeDistance(M) = Temp, and insert M into its position in OPEN according to increasing values of NodeDistance.

Note that although this method, which we shall call "Uniform-cost" is actually blind (there is no evaluation function biasing it to move more quickly toward the goal), it does bear some similarity with best-first search. In fact, the only substantial difference is in the meaning and computation of the node-distance function. In the best-first method, it gave an estimate of a node's proximity to the goal; here it reflects a node's distance from the start node. Besides the difference in where the distance is to or from, there is also a distinct difference in that best-first search uses estimated distances or heuristic ordering values, whereas uniform-cost search uses (in theory) exact distances.

Let us now illustrate the effect on the solution to the French route problem brought about by the use of actual road distances and the uniform-cost algorithm.

```
def unifcost(start, goal):
```

```
    OPEN = [start]                    # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    FVALUE[start] = 0
    while OPEN != []:                 # Step 2
        n = deleteMin(OPEN, f)        # Step 3
        CLOSED.append(n)
        global COUNT
        COUNT += 1
        if n == goal:
            return extractPath(n)
        lst = successors(n)           # Step 4
        lst = listDifference(lst, CLOSED)
        for elt in lst:
            temp = f(elt, n)
            if elt in OPEN:
                if temp < FVALUE[elt]:
                    FVALUE[elt] = temp
                    PREDECESSOR[elt] = n
                    OPEN.remove(elt)
                    insert(elt, OPEN)
            else:
                if not elt in CLOSED:
                    FVALUE[elt] = temp
                    PREDECESSOR[elt] = n
                    OPEN = insert(elt, OPEN)
    # end of loop. This is implicitly # Step 6
```

The supporting functions that are new or have definitions that supersede those used previously are the following.

```
# f computes the distance from the start node to node m
# by adding the distance from n to m to n's distance.
def f(m, n):
    return FVALUE[n] + DISTANCE[m][n]

def deleteMin(lst, fn):
    minVal = 9999
    minElt = None
    for e in lst:
        temp = FVALUE[e]
        if temp < minVal: minVal = temp; minElt = e
    lst.remove(minElt)
    return minElt
```

```
def insert(elt, lst):
    if lst == []: return [elt]
    if FVALUE[elt] < FVALUE[lst[0]]:
        return [elt]+lst
    else: return [lst[0]] + insert(elt, lst[1:])
```

Here's a version of the testing function `test` adapted to `unifcost`.

```
def test():
    start = 'Rennes'
    goal = 'Avignon'
    print "Starting a depth-first graph search from " + start
    path = unifcost(start, goal)
    print path
    global COUNT
    print str(COUNT) + " nodes expanded during the search."
    print 'Total distance is ' + str(FVALUE[goal])
```

We invoke everything by evaluating the call `test()`. The result of applying the uniform-cost method here is the path

```
['Rennes', 'Nantes', 'Limoges', 'Lyon', 'Avignon']
```

with 18 nodes being expanded. As one might expect, the optimal route is different when real distances are used (as just done here) rather than the number of arcs along a path in our particular graph.

5.4.4 The A* Algorithm

If the exact distances from the start node can be determined when nodes are reached, then the uniform-cost procedure can be applied, as we have just seen. When additionally, some heuristic information is available relating the nodes visited to the goal node, a procedure known as the A* (pronounced "Eh star") algorithm is usually better.

The A* algorithm opens nodes in an order that gives highest priority to nodes likely to be on the shortest path from the start node to the goal. To do this it adds g, the cost of the best path found so far between the start node and the current node, to the estimated distance h from the current node to some goal node. Provided that the estimate h never exceeds the true distance between the current node and the goal node, the A* algorithm will always find a shortest path between the start node and the goal node. (This is known as the *admissibility* of the A* algorithm.)

In a sense, the A* technique is really a family of algorithms, all having a common structure. A specific instance of an A* algorithm is obtained by specifying a particular estimation function h.

ALGORITHM A*
begin
input the start node S and the set GOALS of goal nodes;
OPEN $\leftarrow \{S\}$; CLOSED $\leftarrow \phi$;
G[S] $\leftarrow 0$; PRED[S] $\leftarrow$ NULL; *found* $\leftarrow$ **false**;
while OPEN is not empty and *found* is **false do**
begin
L $\leftarrow$ the set of nodes on OPEN for which F is the least;
if L is a singleton **then** let X be its sole element
else if there are any goal nodes in L
then let X be one of them
else let X be any element of L;
remove X from OPEN and put X into CLOSED;
if X is a goal node **then** *found* $\leftarrow$ **true**
else begin
generate the set SUCCESSORS of successors of X;
for each Y **in** SUCCESSORS **do**
if Y is not already on OPEN or on CLOSED **then**
begin
G[Y] $\leftarrow$ G[X] + *distance*(X, Y);
F[Y] $\leftarrow$ G[Y] + $h(Y)$; PRED[Y] $\leftarrow X$;
insert Y on OPEN;
end
else /* Y is on OPEN or on CLOSED */
begin
$Z \leftarrow$ PRED[Y];
temp $\leftarrow$ F[Y]$-$ G[Z] $-$ *distance*(Z, Y)$+$
$+$ G[X] + *distance*(X, Y);
if *temp* $<$ F[Y] **then**
begin
G[Y] $\leftarrow$ G[Y]$-$ F[Y]$+$*temp*;
F[Y] $\leftarrow$*temp*; PRED[Y] $\leftarrow X$;
if Y is on CLOSED **then**
insert Y on OPEN and remove Y from CLOSED;
end;
end;
end;
end;
if *found* is **false then** output "Failure"
else trace the pointers in the PRED fields from X back to S, "CONSing"
each node onto the growing list of nodes to get the path from S to X;
end.

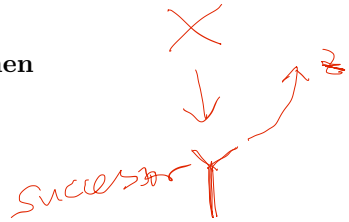

Figure 5.6: A* algorithm. It uses real distances and an estimation function h to efficiently search a state space.

If the estimate h is always zero, A* is certainly admissible. In fact, A* then is no different from the uniform-cost method. In this case the search algorithm is called *uninformed*. The most informed algorithm possible would have h being the exact distance from the current node to the goal. An algorithm A_1 is said to be *more informed* than an algorithm A_2 if the heuristic information of A_1 permits it to compute an estimate h_1 that is everywhere larger than h_2, the estimate computed by A_2.

The assumption that h never exceeds the true distance between the current node and the goal allows admissibility to be assured. An additional constraint called the *consistency assumption* will allow us to assert a kind of optimality for A*. The consistency assumption is satisfied provided that for any two nodes n_1 and n_2, the difference in values of h for those nodes never exceeds the true distance between n_1 and n_2. The A* method is optimal in the sense that if A_1 is more informed than A_2, and the consistency assumption is satisfied, then A_1 never opens any node not opened by A_2.

The general form of the A* algorithm is given in Figure 5.6. Here upper-case nonitalic identifiers represent either fields of nodes or variables whose values are sets of nodes (for example, OPEN, L); italic uppercase letters represent variables whose values are nodes (for example, X); and italic lower-case identifiers represent real-valued or boolean-valued variables (for example, *temp, found*) or functions of nodes or pairs of nodes (for example, h, *distance*). In this formulation, each node X has three data fields associated with it: G[X], the distance from the start node to X along the shortest path found so far; F[X], the sum of G[X] and $h(X)$; and PRED[X], the predecessor of X along the shortest known path to X. It is assumed that *distance*(X, Y) gives the length (or cost) of the arc from X to Y.

For our Python implementation of A*, we need one more hash table to represent the G values.

```
GVALUE = {}
```

We now provide a Python implementation of the A* algorithm. The road distances between cities in kilometers are as for uniform-cost search. The longitudes (in tenths of a degree) of each city are as for best-first search.

```
def AStar(start, goal):
    OPEN = [start]                    # Step 1
    CLOSED = []
    PREDECESSOR[start] = None
    GVALUE[start] = 0
    FVALUE[start] = f(start)
    global COUNT
    COUNT+=1
    while OPEN != []:                 # Step 2
        n = deleteMin(OPEN, f)        # Step 3
```

```
        CLOSED.append(n)
        COUNT += 1
        if n == goal:
            return extractPath(n)
        lst = successors(n)           # Step 4
        lst = listDifference(lst, CLOSED)
        for elt in lst:
            if not ((elt in OPEN) or (elt in CLOSED)):
                GVALUE[elt]=g(elt,n)
                FVALUE[elt]=f(elt)
                OPEN = insert(elt, OPEN)
                PREDECESSOR[elt] = n
            else:
                z = PREDECESSOR[elt]
                if z:
                    temp = FVALUE[elt] - GVALUE[z]\
                                       - arcDist(z,elt) \
                           + GVALUE[n] + arcDist(n,elt)
                else:
                    temp = FVALUE[elt]
                if temp < FVALUE[elt]:
                    GVALUE[elt] = GVALUE[elt] - FVALUE[elt]\
                                              + temp
                    FVALUE[elt] = temp
                    if elt in OPEN:
                        OPEN.remove(elt)
                        OPEN = insert(elt, OPEN)
                    if elt in CLOSED:
                        OPEN = insert(elt, OPEN)
                        CLOSED.remove(elt)
    # end of loop. This is implicitly # Step 6
```

The supporting functions are largely those used previously, with a few modifications. Here are the functions `f`, `g`, and `h` and a new version of `test`, which uses `AStar` to find a path from Rennes to Avignon:

```
def f(n):
    return GVALUE[n] + h(n)

def g(m, n):
    return GVALUE[n] + arcDist(m, n)

def h(n):
    global GOAL
    return 10 * longitudeDiff(n, GOAL)
```

The following function access the hashed distance information, and if the requested entry is not there, it returns an arbitrary very large value.

```
def arcDist(n1, n2):
    try:
        ad = DISTANCE[n1][n2]
    except KeyError:
        ad = 9999
    return ad
```

Here is the version ot `test` adapted for `AStar`.

```
def test():
    start = 'Rennes'
    goal = 'Avignon'
    print "Starting an A* search from " + start
    path = AStar(start, goal)
    print path
    global COUNT
    print str(COUNT) + " nodes expanded during the search."
    print 'Total distance is ' + str(FVALUE[goal])
```

By using ten times the absolute value of the longitude difference to estimate the distance from the current node to the goal, we obtain a more informed algorithm than the uniform-cost method. For the test above, we achieve the same (shortest) route, but we only expand 15 nodes, instead of 18. In a more complicated graph or state space, the savings could well be more dramatic.

5.5 Iterative-Deepening Depth-First Search

As an alternative to depth-first search and breadth-first search one may use "iterative-deepening" depth-first search.

Iterative deepening is a search strategy that often gains the benefits of both breadth-first search and depth-first search without their disadvantages. Breadth-first search has the desirable property that when it finds a goal node, it has found it by a shortest path. Depth-first search, on the other hand, if it reaches a goal node at all, may reach it via a roundabout path. The disadvantage of breadth-first search is that large amounts of memory are typically needed to store the nodes that have been reached but not yet expanded. Depth-first search, on the other hand, only requires memory in proportion to the current depth of the search.

Iterative deepening uses successive depth-first searches to simulate a breadth-first search. Starting each time from the initial state, it does limited-depth depth-first searches for depths 1, 2, 3, etc., until one of these searches finds a goal node. Iterative deepening is guaranteed to find any goal node by a shortest path, just

like breadth-first search, because it always explores all paths of length $n-1$ before paths of length n. Iterative deepening requires memory only proportional to the current depth of search, because it is implemented as a series of depth-first searches.

It might appear at first that iterative deepening is an inefficient way to search, because many of the nodes are processed repeatedly on successive iterations. However, because there are relatively few nodes close to the initial node, when compared with the number far away, the repeated searching takes relatively little of the overall searching time. Empirical studies have shown that iterative-deepening searches are often more time-efficient than breadth-first search, because the lessened memory requirements lead to reduced overhead for accessing state information.

To quantify the effect of the repeated searching of some nodes, let us assume that a state-space search problem has a branching factor of b and that a closest goal node lies at distance d from the start node. Breadth-first search makes a number of node visits equal to

$$\sum_{i=0}^{d} b^i$$

which is approximately equal to $b^{d+1}/(b-1)$. The iterative-deepening search uses

$$\sum_{i=0}^{d} \sum_{j=0}^{i} b^j$$

node visits. As for breadth-first search, the quantity is $O(b^d)$; this is easy to see because the quantity attributable to the last iteration is $O(b^d)$, the second to last iteration is $O(b^{d-1})$, etc., and the sum of these is $O(b^d)$. As b gets larger, the ratio of numbers of nodes visited by iterative-deepening depth-first search and breadth-first search approaches $(1+b)/b$, which in turn approaches unity. Thus iterative-deepening depth-first search uses relatively few additional node visits over the number made by breadth-first search.

5.6 Genetic Search Methods

5.6.1 An Evolution Metaphor

Because of the combinatorial explosion in most of the interesting kinds of search problems, researchers have been looking for good heuristics for guiding search. One approach that has become increasingly popular in recent years is to create a "population" of candidate solutions to a problem and define some mutation, crossover, and selection methods to permit this population to evolve in the hope that a true solution to the problem will appear in the population at some point.

Thus the approach is to model Darwinian evolution and create a sort of "environment" where the fittest individuals survive and the weakest tend to be eliminated.

One of the attractions of this approach is that the evolution can be biased to move in the direction of the goal through the use of good measures of "fitness" and appropriate criteria for crossovers' and mutations' taking place.

Another attraction is that this approach is easily amenable to parallel processing, with the potential to reduce search times provided enough computer processors are available. Parallel processing being a rapidly developing technology, this has some promise in making complex search problems more manageable in practice.

5.6.2 Terminology

The candidate solutions or individuals in the population are sometimes referred to as "chromosomes," since they are characterized by strings or lists analogous to the genetic information encoded by DNA molecules.

A new chromosome can be formed from an old one by "mutation" whereby a relatively small change is made to the old one, often randomly. Another way a new chromosome can be formed is by the "crossing" of two existing chromosomes; each of two existing chromosomes is split at some dividing point (sometimes chosen randomly) and the pieces are rearranged to form two new chromosomes, each having a piece from one of the original chromosomes.

5.7 A Genetic Algorithm in Python

We illustrate the genetic algorithm paradigm using a well-known combinatorial optimization problem, the traveling salesman problem.

5.7.1 The Traveling Salesman Problem

In an instance of the traveling salesman problem (TSP), one is given a set of n cities and the $n(n-1)/2$ pairwise distances between them. The goal is to find a minimum-length path that visits each city exactly once except that it returns to the starting city. A closed path that visits every city exactly once is a *tour*.

5.7.2 Crossover and Mutation of Paths

In order to apply the genetic algorithm approach, we must identify an appropriate class of individuals, instances of which can be members of the changing population. This class must include any solutions to our problem, if solutions exist. For the traveling salesman problem, let an individual be a path in the graph of cities. A path can be represented as a sequence of cities.

We must also define an appropriate set of operators that perform transformations on individuals to obtain new ones. Our operators will produce new paths as follows. The crossover of paths `path1` and `path2` at position k is obtained by dividing `path1` at position k into left and right parts (let's call them LEFT1 and RIGHT1) and similarly dividing `path2` (into LEFT2 and RIGHT2). There are theoretically two new paths produced by crossover: the result of appending LEFT1 with RIGHT2 and the result of appending LEFT2 with RIGHT1. In our program, we will take only the first of these. Assuming that `path1` and `path2` are chosen at random, either of the two results is equally likely. (Note: these are called "one-point" crossovers. Two-point and other sorts of crossovers can also be used and may improve performance.)

Two kinds of mutation are defined: `mutate1` produces a variant of a path in which one randomly selected city in the path is replaced by another randomly selected city. The other kind of mutation, produced by `mutate2`, selects two positions in the `path` and exchanges the cities at those positions. This has the property of always producing a tour if the input path is a tour.

5.7.3 The `evolve` Function

The evolutionary process can be described with a program loop that updates the population over and over again, until some termination condition is reached (or a fixed number of generations have been created):

```
Procedure Evolve:
  set up an initial population;
  begin loop (for g generations):
    Create m mutations and add them to the population,
     subject to population limits and their relative fitness.
    Create c crossovers and add them to the population,
     subject to population limits and their relative fitness.
  end loop.
```

5.7.4 The Program

Here is an implementation of our genetic algorithm in Python. It begins with expressions that build the representation of the network of cities. Then it provides functions that implement the crossover and mutation operators and the evolutionary search process itself.

```
CITIES = ['Seattle','Portland','Spokane','Wenatchee','Bellingham']
NCITIES = len(CITIES)

DISTANCE = {}
DISTANCE['Seattle']    = {'Portland':150,'Spokane':350,\
```

```
                          'Wenatchee':100,'Bellingham':90}
DISTANCE['Portland']   = {'Seattle':150,'Spokane':400,\
                          'Wenatchee':200,'Bellingham':235}
DISTANCE['Spokane']    = {'Portland':400,'Seattle':350,\
                          'Wenatchee':275,'Bellingham':385}
DISTANCE['Wenatchee']  = {'Portland':200,'Spokane':275,\
                          'Seattle':100,'Bellingham':130}
DISTANCE['Bellingham'] = {'Portland':235,'Seattle':90,\
                          'Spokane':385,'Wenatchee':130}
```

We represent an individual as a two-element list whose left part is a list of cities and whose right part is a strength value.

```
def getPath(individual):
    return individual[0]

def getStrength(individual):
    return individual[1]
```

The initial population is just a single individual containing a bad "solution" to our problem with a very weak strength:

```
INITIAL_POPULATION =\
  [[['Seattle','Seattle','Seattle','Seattle','Seattle'], 0]]
```

The population, current minimum strength, and current population size vary during a run, and we represent them with global variables.

```
POPULATION = INITIAL_POPULATION
(defvar *current-min-strength*) ; initially 0
(defvar *current-pop-size*) ; initially 1
```

The maximum number of individuals permitted in the population is a parameter that affects both the running time and the effectiveness of the evolutionary process. The smaller the population, the greater the chances of getting stuck in a local maximum of the strength function.

```
POPULATION_LIMIT = 16
```

Here we define functions that help us evaluate a list of cities as a potential tour route.

```
def arcDist(c1, c2):
    if c1 == c2: return 0
    try:
        ad = DISTANCE[c1][c2]
    except KeyError:
```

```
        ad = DISTANCE[c2][c1]
    return ad

def cycleCost(path):
    return arcDist(path[0],path[-1]) + pathCost(path)

def pathCost(path):
    if len(path) < 2: return 0
    return arcDist(path[0],path[1]) + pathCost(path[1:])

def nonTourPenalty(path):
    return 100 * (len(setDifference(path, CITIES)) +\
                  len(setDifference(CITIES, path)))

def setDifference(s1, s2):
    s = s1[:]
    for elt in s2:
        if elt in s:
            s.remove(elt)
    return s

def chromosomeStrength(individual):
    return 10000.0 / \
        (1 +2*cycleCost(individual[0]) +\
         50*nonTourPenalty(individual[0]))
```

Now we define functions that produce variations of individuals in the population.

```
def mutate(individual):
    if choice([0,1]) == 0:
        return mutate1(individual)
    else:
        return mutate2(individual)

def mutate1(individual):
    where = choice(range(NCITIES))
    newCity = choice(CITIES)
    newIndiv = [individual[0][:], individual[1]]
    newIndiv[0][where]=newCity
    return newIndiv

def mutate2(individual):
    where1 = choice(range(NCITIES))
    where2 = choice(range(NCITIES))
    city1 = individual[0][where1]
```

```
    city2 = individual[0][where2]
    newIndiv = [individual[0][:], individual[1]]
    newIndiv[0][where1]=city2
    newIndiv[0][where2]=city1
    return newIndiv
```

The function `crossover` implements a form of hybridization; it combines parts of two individuals to form a new one.

```
def crossover(individual1, individual2):
    where = choice(range(NCITIES))
    newIndiv1 = individual1[:where] + individual2[where:]
    #newIndiv2 = individual2[:where] + individual1[where:]
    return newIndiv1
```

Here is the main function for our genetic search algorithm. The function `evolve` accepts three parameters that specify the number of generations desired and how many mutations and crossovers to try in each generation.

```
def evolve(ngenerations, nmutations, ncrossovers):
    global POPULATION; global INITIAL_POPULATION;
    POPULATION = INITIAL_POPULATION
    for i in range(ngenerations):
        for j in range(nmutations):
            mutant = mutate(choice(POPULATION))
            print 'mutant is ' + str(mutant)
            addIndividual(mutant)
        for j in range(ncrossovers):
            theCross =\
                crossover(choice(POPULATION), choice(POPULATION))
            print 'theCross is ' + str(theCross)
            addIndividual(theCross)
        print 'In generation ' + str(i) + ', the population is: '
        print str(POPULATION)
```

The function `addIndividual` takes care of the business of determining whether a new individual survives by being inserted into the POPULATION list.

```
def addIndividual(individual):
    global POPULATION; global POPULATION_LIMIT
    strength = chromosomeStrength(individual)
    individual[1] = strength
    if len(POPULATION) < POPULATION_LIMIT:
        POPULATION.append(individual)
    else:
        deleteMin()
        POPULATION.append(individual)
```

Here is a helping function for `addIndividual`.

```
def deleteMin():
    minSoFar = 9999
    for elt in POPULATION:
        if elt[1] < minSoFar:
            minSoFar = elt[1]
            theIndiv = elt
    POPULATION.remove(theIndiv)
```

The function `test` sets up a demonstration run of the `evolve` function, specifying 10 generations, 10 mutations per generation, and 10 crossovers per generation.

```
def test()
    evolve(10, 10, 10) )
   # These values often lead
   # to convergence at strength 4.78.
```

Note that the results of each run are generally different, due to the use of random numbers in the process of evolution. Most of the time, an instance of the optimal solution is a member of the final population. However, sometimes the program gets stuck in a local minimum or fails to converge within 10 generations.

5.7.5 Discussion

An important question for genetic algorithms is this: Under what conditions do they find an optimal solution? In other words, can we design a system so that it is guaranteed to produce a most fit individual, and if so, how? There are several things that must be done correctly for things to work reliably. Let us first consider the behavior of the program we have.

The results obtained with this algorithm are not entirely predictable, due to the random choices made in each run. If there are not enough mutations in relation to the number of crossovers, the search can easily get stuck in local maxima. The probability of finding an optimal tour can be increased by changing the parameters to `evolve` so as to allow more mutations and fewer crossovers.

There may also be a tendency for the population to become homogeneous. To counteract this, more emphasis could be placed on random mutations of type 1.

An explicit mechanism to encourage diversity could lead to a richer gene pool and potentially lower-cost tours. Implementing such a mechanism is one of the suggested exercises.

In order to have a genetic algorithm that reliably finds an optimal individual, we must have a means of producing new individuals that is capable of transforming some member of the initial population to the desired individual; i.e., the goal must be "reachable" under the mutation and crossover transformations. It is also

necessary that each ancestor of the goal along one of the possible transformation sequences leading to the goal be permitted to join the population and survive long enough to spawn its successors that are also in the sequence. Thus it is not enough that the goal be reachable by application of the transformations, because if early potential ancestors of the goal are not themselves fit enough, they will be eliminated before they have a chance to spawn.

Thus it may be beneficial to avoid a strict fitness criterion for survival. Some probabilistic mechanism that takes fitness into some account, but does not rely totally on it, might thus be better as a population control device.

As long as there is the mathematical possibility of reaching the goal through an evolutionary process, the probability of finding a goal can be increased by performing multiple runs of the whole process.

The genetic algorithm approach to search is related to an optimization technique known as "simulated annealing." Both methods involve the use of biased random transitions from one state to another. The genetic approach maintains a whole population of individuals that gradually evolves toward a population containing a solution, while simulated annealing generally maintains only a single candidate solution at any one time. However, the single candidate in simulated annealing may evolve through a much larger number of states than the genetic algorithm has generations. The single candidate is typically a large, structured object with many components (e.g., pixels of an image), all of which typically change a little bit from one iteration of simulated annealing to the next.

Genetic algorithms are sometimes thought to perform learning, and the genetic-algorithm method is sometimes classified as a learning method rather than a search method. (For more on learning, see Chapter 10). As a search method, the genetic algorithm technique illustrates how randomness can be exploited to spread the exploration within the state space.

5.8 Two-Person Zero-Sum Games

In games like checkers, chess, go, and tic-tac-toe, two players are involved at a time. If player A wins, then B loses, etc. Such a game is called a two-person, zero-sum game. The fact that a gain by one player is equivalent to a loss by the other leads to a sum of zero overall advantage.

The state-space graphs for two-person, zero-sum games are generally regarded differently from other state-space graphs. From any particular position in the game (that is, a state), at issue are the possible moves one or one's opponent can make and what the consequences of each move may be.

In tic-tac-toe, one must try to get three X's in a line, or three O's in a line, while preventing one's opponent from attaining such a line. One may settle for a draw, alternatively. The states from which no further move can be made may be called *final states*. The final states fall into three categories: win for X, win for O, or draw. When a win is impossible or unlikely, the objective may be to

attain a draw. By assigning values to the possible states, the problem of playing the game becomes one of trying to maximize or minimize the value of the final state. Let us now examine a procedure for this.

5.8.1 Minimaxing

For a player, finding a path from the current state to a goal state (or a state with high value) is not an adequate basis on which to choose a move, if the opponent's actions also affect the outcome. Generally speaking, the opponent does not cooperate; it is necessary to take the opponent's probable reactions into account in choosing a move.

The set of states reachable from a given state (using only legal moves of a game) may be arranged into a *game tree*. If a particular state is reachable along two or more alternative paths, we may replicate the state enough times to allow a tree rather than a general graph to represent the current set of game potentialities ("If my opponent goes there, then I could go here," etc.). Let us call the two players "Max" and "Min." We can generally assign a value to each possible state of a game in such a way that Max desires to maximize that value and Min wants to minimize it. Such an assignment of values is a kind of evaluation function. In games whose states are defined by the placements of pieces on boards (like checkers and chess), such an assignment is often called a "board evaluation function."

For the game of tic-tac-toe, a board evaluation function is $100A + 10B + C - (100D + 10E + F)$ where A is the number of lines of three X's, B is the number of unblocked lines with a pair of X's, and C is the number of unblocked lines with a single X. Similarly D, E, and F give numbers of lines of O's in various configurations.

A game tree for tic-tac-toe is shown in Figure 5.7. Each level of nodes in the tree is called a *ply*. Ply 0 contains only a single node, corresponding to the current board position. Ply 1 contains the children of the root of the tree. Typically, a game-playing program will generate all the board positions for nodes down to a particular ply such as 4. It will then evaluate the leaves (tip nodes) of that four-level tree with the board evaluation function and obtain what are called *static* values. Then a process of backing values up the tree is begun. If the root corresponds to a position where it is Max's move, then all even-numbered ply contain "Max nodes" and all odd-numbered ply contain "Min nodes." To back up the value to a Max node, the maximum child value is written into the node. For a Min node, the minimum child value is taken. Backing-up proceeds from the leaf nodes up, until the root gets a value. Max's best move (given, say, a 4-ply analysis) is the move that leads to the largest value at a leaf node, given that Max always will maximize over backed-up values and that Min will always minimize. The value thus associated with each nonleaf node of the tree is the node's "backed-up value." While the backed-up value for a node is being computed and after the backed-up value for the node's first descendant has been

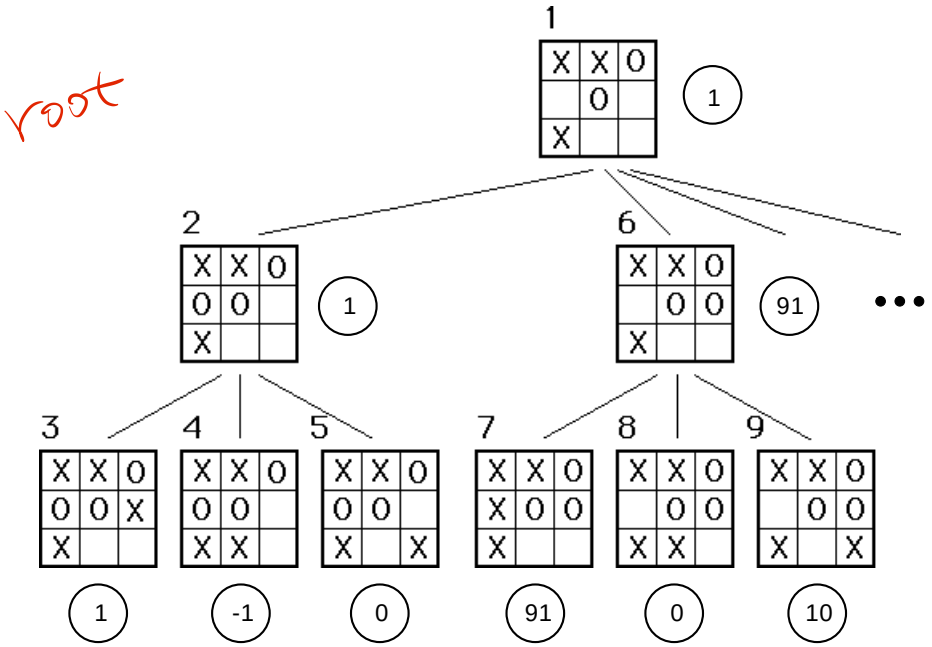

Figure 5.7: Game tree for tic-tac-toe.

computed, the node has a "provisional backed-up value," which is the current minimum (or maximum, depending on whose move it is at the node). The positions shown in Figure 5.7 are numbered according to the order in which they would be generated. The six leaf nodes are labeled with their static values. Nodes 2 and 6 are labeled with their backed-up values. Since the root corresponds to a position where O, the minimizing player, is to move, choosing between positions 2 and 6, it is clear that O prefers 2 with value 1 over position 6 with a value of 91.

5.8.2 AND/OR Graphs

Game trees with Min and Max nodes have a counterpart in problem solving; they are called AND/OR trees and, more generally, AND/OR graphs. An AND/OR tree expresses the decomposition of a problem into subproblems, and it allows alternative solutions to problems and subproblems. The original problem corresponds to the root of the AND/OR tree. At an AND node, all the child nodes must be solved in order to have a solution for the AND node. At an OR node, at least one of the children must be solved, but not necessarily any more than one. An AND/OR tree is illustrated in Figure 5.8. The overall goal for this example is to prepare a main course for a dinner. Disjunctive subgoals of the main goal are

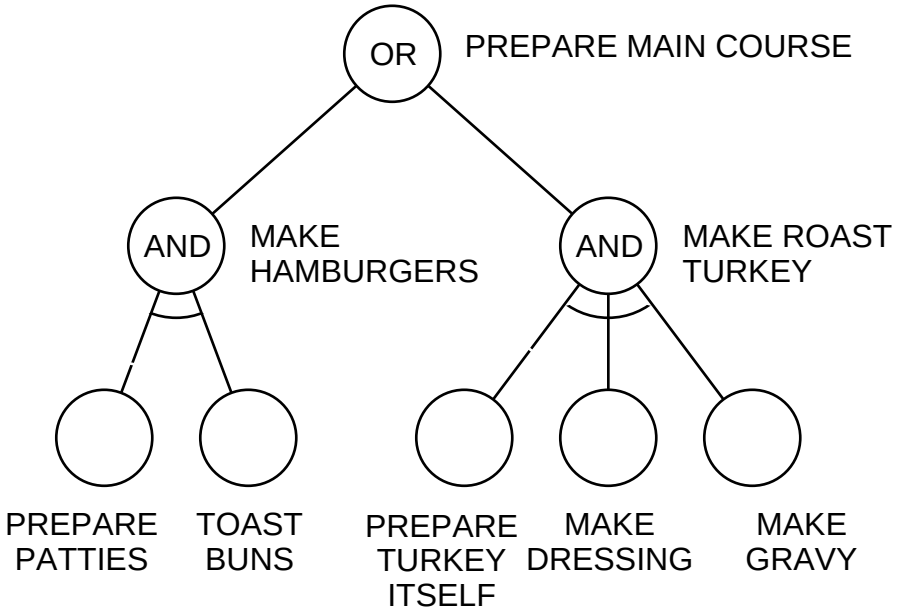

Figure 5.8: An AND/OR tree.

"make hamburgers" and "make roast turkey." Conjunctive subgoals of "make hamburgers" are "prepare patties" and "toast buns."

In an AND/OR tree (or an AND/OR graph), a node is *solved* if

- It is an OR node and at least one of its children is solved.
- It is an AND node and all its children are solved.
- It is a leaf node and problem-dependent criteria associated with the node are satisfied.

In this example, the problem-dependent criteria are "patties prepared," "buns toasted," and so on.

A *solution graph* for an AND/OR graph (or tree) is a subgraph of the original consisting of a set of solved nodes and the arcs connecting them that make the root node solved. The root of the original graph is a necessary part of a solution graph. For the example of Figure 5.8, there are two solution graphs; one of these consists of the nodes PREPARE MAIN COURSE, MAKE HAMBURGERS, PREPARE PATTIES, and TOAST BUNS, together with the arcs that connect these nodes. The *solution* to the problem (or to any subproblem) may be defined recursively: if the (sub)problem corresponds to a leaf node N, then its solution

is given by the satisfaction of the appropriate problem-dependent criteria; otherwise, the (sub)problem corresponds to an interior node N, and its solution consists of the subproblems that correspond to the children of N contained in the solution graph, together with their solutions. In this example, the solution may be considered to be a set of plans one of which is simply MAKE HAMBURGERS and another of which is PREPARE PATTIES *and* TOAST BUNS.

An AND/OR graph may sometimes be used to represent the state space for a game. For example, in a game of tic-tac-toe, we may consider the objective to be solving a problem; that problem is to win the game. For the X player, the problem is clearly solved if the current position contains three X's in a row and does not contain three O's in a row. Any position is solved if X has a win at it or can force a win from it, and otherwise, it is not solved. If it is X's move, X can force a win if *any* of the successor positions to the current one is solved. If it is O's move, then X can force a win only if *all* the successors of the current position are solved. In order to use an AND/OR graph as a basis for playing the game, it must be computationally feasible to generate the entire graph for the current position. Consequently, AND/OR graphs are usually of use only for relatively simple games, for which the number of moves in a game can never get larger than, say, 15 or 20. On the other hand, the minimaxing approach can handle more complex games, because it allows the use of heuristic evaluation functions that avoid the necessity of constructing the entire game tree.

5.8.3 Alpha-Beta Search

In order to play well, a program should examine alternative lines of play to ply as deep as possible. Unfortunately, the number of possible board positions in a game tree tends to grow exponentially with the number of ply. It is usually possible to prune off subtrees as irrelevant by looking at their roots in comparison with alternative moves at that level. For example, in chess, one may be examining a line of play in which it suddenly is discovered that the opponent could capture the queen, while nothing would be gained and a better move is available. Then there is no point to examining alternatives in which the opponent kindly does not take the queen. By assumption, each player is trying to win: one by maximizing and one by minimizing the evaluation function value. When such an irrelevant subtree is discovered, it is generally called a "cutoff." A well-known method for detecting cutoffs automatically is the "alpha-beta" method.

Let us consider an example in a game of checkers. We assume that the game has progressed to the position shown in Figure 5.9, with Black to move. The 3-ply game tree for this position is shown in Figure 5.10. The moves are examined in an order that considers pieces at lower-numbered squares first and, for alternative moves with the same pieces, moves to lowest-numbered squares first. Black has four possible moves from the position shown, and so there are four branches out from the root of the tree. The first move to be considered is to move the piece at square 20 to square 16. To this, Red has six possible

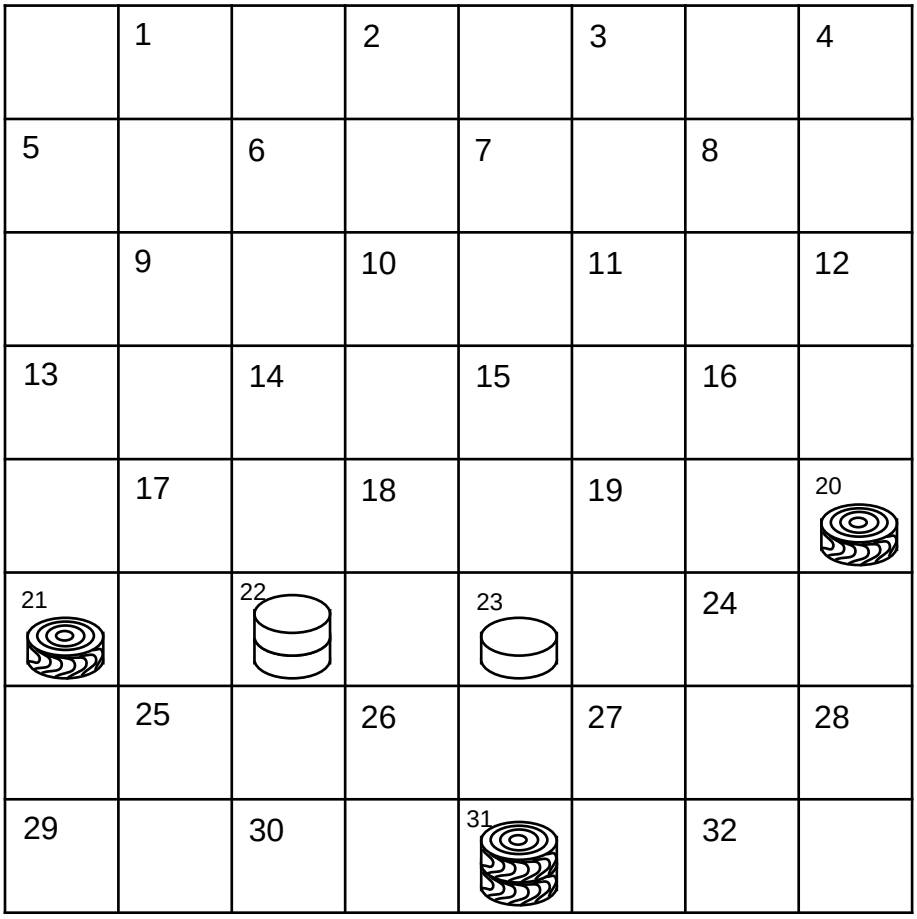

Figure 5.9: Checkers position with Black to move.

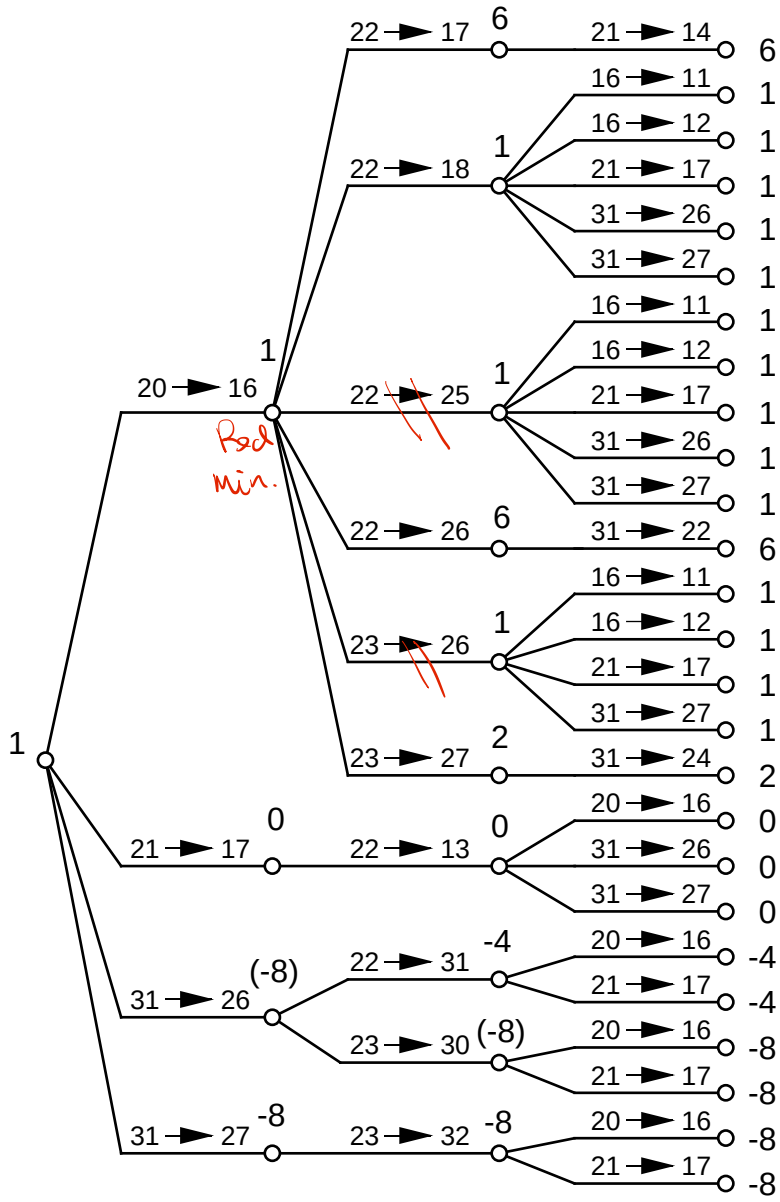

Figure 5.10: A three-ply game tree for the checkers position.

replies, the first of which is $22 \rightarrow 17$. In this case, Black must then jump with $21 \rightarrow 14$. This position is in ply 3 and should be evaluated statically. We use the evaluation function

$$5x_1 + x_2$$

Here x_1 represents Black's king advantage (the number of black kings minus the number of red kings), while x_2 represents Black's single-piece advantage.

The static value of the first ply-3 position is therefore 6. This value is backed up to the previous node, and it also becomes the provisional backed-up value for the predecessor of that node (the first descendant of the root). The second Red reply to Black's $20 \rightarrow 16$ is now considered; $22 \rightarrow 18$ leads to five alternatives for Black, each of which results in some position with static value 1. The maximum of these is, of course, also 1, and this value is the backed-up value of the position resulting from Red's second alternative. This value, 1, is less than the provisional backed-up value (6) of the predecessor node, and since Red is minimizing, replaces 1 as the provisional value.

Red's next alternative is $22 \rightarrow 25$. Black's first response, $16 \rightarrow 11$, gives a ply-3 value of 1. Since this is at least as bad for Red as any consequence of Red's previous alternative $22 \rightarrow 18$, if any of Black's other responses to Red's $22 \rightarrow 25$ were better for Black, they would be even worse for Red, and Red would avoid $22 \rightarrow 25$. Thus Black's search can ignore all further possible responses to Red's $22 \rightarrow 25$. Here we say that a *beta cutoff* occurs. Another, similar beta cutoff occurs for Red's alternative $23 \rightarrow 26$.

Red's six alternatives to Black's first move lead to backed-up values of 1, 1, 1, 6, 1, and 2. Since none of the last five is less than the current provisional backed-up value of 1, the final backed-up value for the first descendant of the root is 1.

Black's second alternative, $21 \rightarrow 17$, leads to a position with backed-up value 0. Since Black is maximizing, this move is clearly inferior to the first.

Black's third alternative, $31 \rightarrow 26$, brings out another kind of cutoff. Red's first alternative for a reply gives Black two choices, each leading to a static value of -4. Thus Red can force Black into a situation much less favorable than if Black were to choose the first or second move in the first place. In other words, after computing the backed-up value of the position after Red's first (and in this case, only) alternative, $22 \rightarrow 31$, this value becomes the provisional value of the preceding position, but here a comparison is made: if this provisional value is less than any of the backed-up values already determined for this ply, the move is certainly inferior, and further evaluation of positions in this subtree can be bypassed. We say here that an *alpha cutoff* occurs at this node. Were the other positions of this subtree to be evaluated, the result would be that the backed-up value of the position after $31 \rightarrow 26$ is -8, which is even worse than the provisional value of -4. But, even if the other positions had high values, the effort to evaluate these positions would be wasted, since Red would always choose the alternative least favorable to Black.

Black's fourth alternative (the last) is $31 \rightarrow 27$. This leads to ply-3 positions of value -8 and so makes this move inferior. Black's best move is clearly the first, according to the 3-ply analysis we have performed.

We saw how beta and alpha cutoffs were used in analyzing Black's first and third alternatives, respectively, to avoid evaluating some of the positions in the full 3-ply tree. Such cutoffs can be systematically determined with the *alpha-beta pruning* procedure. Alpha cutoffs are used at minimizing levels, while beta cutoffs are used at maximizing levels. Let A be a maximizing-level node for which k alternatives have been tried, the maximum backed-up value of these being alpha. Let B be the minimizing-level node that is the result of the next alternative from A. As soon as any immediate descendant of B receives a backed-up value that is less than alpha, further consideration of the subtree at B is unnecessary (an alpha cutoff occurs).

Similarly, let C be a minimizing-level node for which k alternatives have been tried, the minimum backed-up value of these being beta. Let D be the maximizing-level node that is the result of the next alternative from C. As soon as any immediate descendant of D receives a backed-up value that is more than beta, the remaining positions in the subtree at D can be ignored (a beta cutoff occurs).

More concisely, an alpha (beta) cutoff occurs at a maximizing (minimizing) node when it is known that the maximizing (minimizing) player has a move that results in a value α (β) and, subsequently, when an alternative to that move is explored, it is found that the alternative gives the opponent the option of moving to a lower (higher) valued position. (Any further exploration of the alternative can be canceled.)

If the search tree is explored in an order that improves the likelihood of cutoffs, the alpha-beta pruning procedure can typically eliminate more than half the nodes that would have to be evaluated in checkers and chess situations. One way to increase the chances of getting useful cutoffs is to apply the evaluation function to each of the ply-1 positions and to order the exploration of the corresponding subtrees in a best-first fashion.

5.9 Bibliographical Information

Depth-first, backtracking search was used extensively in early problem-solving systems such as GPS ("General Problem Solver") [Newell et al. 1959]. Heuristic search techniques were studied by Pohl [Pohl 1969] and E. Sandewall [Sandewall 1969]. AND/OR trees and graphs were studied by J. Slagle and J. Dixon ([Slagle 1963] and [Slagle and Dixon 1969]). A thorough presentation of problem representation and state-space search, including proofs for the admissibility and the optimality of the A* algorithm, may be found in [Nilsson 1971]. Section 5.4.3 on pruned, best-first search is based on material from [Nilsson 1971] and from [Barr and Feigenbaum 1981]. A good source of heuristics for mathematical problem

solving is [Polya 1957]. A book that combines an introduction to state-space search with research results on heuristics is [Pearl 1984]. A collection of papers giving relatively recent research results on search is [Kanal and Kumar 1988], and a good survey article on search techniques is [Korf 1988].

The notion of evolution as a process to produce intelligence is a variation on the theory of the origin of species of Charles Darwin [Darwin 1859]. The development of the genetic algorithm methodology is credited largely to John Holland at the University of Michigan [Holland 1975]. In addition to making his own contributions, he influenced others to develop and apply the genetic algorithm approach. For example, the methodology was applied to automatic configuration of image processing methods [Gillies 1985]. One of the problems in avoiding local optima with genetic search is maintaining a rich pool of genes in the population; this is discussed in [Mauldin 1984] and is the subject of one of the exercises below. The use of the genetic approach in solving traveling salesman problems was discussed in [Goldberg and Lingle 1985] as well as [Brady 1985]. One of the problems in parallelizing a genetic algorithm is handling the survival decision in parallel; this is discussed for the traveling salesman problem in [Suh and Van Gucht 1987]. Among many other applications of genetic algorithms is the automatic design of integrated circuits [Shahookar and Mazumder 1990] and the production of original art [Sims 1991]. For a very readable and comprehensive introduction to genetic algorithms, see [Goldberg 1989]. The two volumes [Koza 1992] and [Koza 1994] constitute a massive monograph on the use of genetic algorithms for automatic programming.

A successful checkers-playing program was developed in the late 1950s [Samuel 1959], and chess has received substantial attention since then ([Newborn 1975], [Berliner 1978], and [Ebeling 1987]). Efforts have also been made to computerize the playing of backgammon [Berliner 1980] and go [Reitman and Wilcox 1979]. Go is considerably different from chess and checkers in that the game trees for it are so wide (there are typically hundreds of alternatives from each position) that normal minimax search is impractical for it. It has been suggested that principles of perceptual organization be incorporated into go-playing programs [Zobrist 1970]. The use of plans in playing chess is described in [Wilkins 1980]. The efficiency of alpha-beta search is examined in [Knuth and Moore 1975].

References

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1983. *Data Structures and Algorithms*. Reading, MA: Addison-Wesley.

2. Barr, A., and Feigenbaum, E. A. (eds.) 1981. *The Handbook of Artificial Intelligence*. Los Altos, CA: William Kaufman.

3. Berliner, H. J. 1978. A chronology of computer chess and its literature.

Artificial Intelligence, Vol. 10, No. 2, April, pp. 201-214.

4. Berliner, H. J. 1980. Computer backgammon, *Scientific American*, June.

5. Brady, R. M. 1985. Optimization strategies gleaned from biological evolution (Letter to the Editor). *Nature*, Vol. 317, pp. 804-806.

6. Darwin, C. 1859. *The Origin of Species.* New York: New American Library, Mentor Paperback.

7. DeJong, Kenneth A., and Spears, W. 1992. A formal analysis of multipoint crossover in genetic algorithms. *Annals of Mathematics and Artificial Intelligence*, Vol. 5, No. 1, pp. 1-26.

8. Ebeling, C. 1987. *All the Right Moves.* Cambridge, MA: MIT Press.

9. Gillies, A. M. 1985. Machine learning procedures for generating image domain feature detectors. Ph.D. dissertation, University of Michigan, Ann Arbor.

10. Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley.

11. Goldberg, David E. and Lingle, R. 1985. Alleles, loci and the travelling salesman problem. *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pp. 154-159.

12. Haralick, R. M., and Elliott, G. L. 1980. Increasing tree-search efficiency for constraint-satisfaction problems. *Artificial Intelligence*, Vol. 14, No. 3, pp. 263-313.

13. Holland, John. 1975. *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press.

14. Kanal, L., and Kumar, V. (eds.) 1988. *Search in Artificial Intelligence.* New York: Springer-Verlag.

15. Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp293-327.

16. Korf, R. E. 1988. Search: A survey of recent results. In Shrobe, H. E. (ed.), *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, San Mateo, CA: Morgan Kaufman, pp. 197-237.

17. Koza, J. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

18. Koza, J. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

19. Mauldin, J. L. 1984. Maintaining diversity in genetic search. *Proceedings of AAAI'84*, pp. 247-250.

20. Newborn, M. 1975. *Computer Chess*. New York: Academic Press.

21. Newell, A., Shaw, J., and Simon, H. 1969. Report on a general problem-solving program. *Proceedings of the International Conference on Information Processing*, Paris, pp. 256-264.

22. Newell, A., and Simon, H. A. 1963. GPS, a program that simulates human thought. In Feigenbaum, E. A., and Feldman, J. (eds.), *Computers and Thought*. New York: McGraw-Hill, pp. 279-293.

23. Nilsson, N. J. 1971. *Problem Solving Methods in Artificial Intelligence*. New York: McGraw-Hill.

24. Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley.

25. Pohl, I. 1969. Bidirectional and heuristic search in path problems. Ph.D. dissertation, Dept. of Computer Science, Stanford University, Stanford, CA.

26. Polya, G. 1957. *How to Solve It*, 2d ed. Garden City, NY: Doubleday.

27. Reitman, W., and Wilcox, B. 1979. The structure and performance of the Interim.2 Go program. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Tokyo, pp, 711-719.

28. Sandewall, E. 1969. Concepts and methods for heuristic search. *Proceedings of the First International Joint Conference on Artificial Intelligence*, Washington, DC, pp. 199-218.

29. Shahookar, K., and Mazumder, P. 1990. GASP: A genetic algorithm for standard cell placement. *Proceedings of EDAC'90, the European Design Automation Conference*, Glasgow, Scotland, UK. Washington, DC: IEEE Computer Society Press, pp. 660-664.

30. Sims, K. 1991. Artificial evolution for computer graphics. *Computer Graphics*, Vol. 25, No. 4 (July), pp. 319-328.

31. Slagle, J. 1963. A heuristic program that solves symbolic integration problems in freshman calculus, *Journal of the ACM*, Vol. 10, No. 4, pp. 507-520.

32. Slagle, J., and Dixon, J. 1969. Experiments with some programs that search game trees. *Journal of the ACM*, Vol. 16, No. 2, pp. 189-207.

33. Suh, J. Y., and Van Gucht, D. 1987. Distributed genetic algorithms. Technical Report 225. Bloomington: Department of Computer Science, University of Indiana,

34. Wilkins, D. E. 1980. Using patterns and plans in chess. *Artificial Intelligence*, Vol. 14, pp. 165-203.

35. Zobrist, A. L. 1970. Feature extraction and representation for pattern recognition and the game of Go. Ph.D. dissertation, Dept. of Computer Science, University of Wisconsin, Madison.

Exercises

1. Define the following terms:

   (a) state space

   (b) goal state

   (c) move generator

   (d) heuristic

   (e) backtrack search

2. Brute-force approaches to searching can get swamped even by "toy" problems.

   (a) How many distinct states are there in the state space for the version of the painted squares puzzle shown in Figure 5.1?

   (b) What are the maximum and minimum numbers of states that there might be in a 4-square case of the puzzle?

   (c) Suppose that each square in Figure 5.1 can be flipped over and that each side is painted with the same pattern on both faces of the square. What is the number of distinct arrangements of the four pieces in the 2×2 space?

   (d) Let us define a "quadramino" as any arrangement of four of the squares of a square tiling of the plane, such that the four form a single connected group. Two such arrangements that are alike except for a translational displacement will be considered as equivalent. If the four pieces of Figure 5.1 may be placed in any quadramino arrangement and the pieces may be flipped, what is the total number of distinct arrangements of the four squares?

3. What is the maximum distance (where distance is measured as the number of arcs in the shortest path between two nodes) between the initial state and a goal state, in a 4-square version of the painted squares puzzle (assuming that there *is* a goal state for the version!).

4. The way in which the squares are painted in a version of the painted squares puzzle apparently affects not only the number of solutions that exist but the efficiency with which the backtracking algorithm finds a solution. Give an example of a version of the puzzle in which many solutions exist but for which a lot of backtracking is required.

5. The solution to the painted squares puzzle in the text suffers from an inefficiency: pieces already placed are repeatedly rotated by the `orient` function. Implement a version of the solution that avoids this redundancy. Determine experimentally how many calls to `rotateList` are used by the old and new versions.

6. Suppose that the rules for the painted squares puzzles are changed, so that (a) a starting configuration consists of all the squares placed to fill the rectangular space, but not necessarily with their sides matching and (b) two kinds of moves are allowed: (i) the rotation of a piece clockwise 90 degrees and (ii) interchanging a piece with an adjacent piece. What is the maximum number of states that there could be in the state space for an instance of this puzzle having a 4 by 4 rectangle (16 squares)? What is the minimum number of states that there might be?

7. A kind of puzzle called an "Equilibrium" puzzle is shown in Figure 5.11. Each piece has four quadrants in which may appear images of earth, air, fire, or water. The object is to place all the pieces to form a rectangle such that each interior corner (where four pieces come together) has precisely one instance each of the four "elements." At exterior corners (where only two pieces meet), the two adjacent elements must not be the same. Like the painted squares puzzle, the number of possible solutions in an instance of Equilibrium depends on how the pieces are painted. Write a program that accepts a description of a set of equilibrium pieces and finds all the solutions for that instance of the puzzle.

8. In Chapter 4, it was shown how knowledge about objects could be organized in an inclusion hierarchy. Suppose that it is desired to build a more knowledge-based solver for the painted squares puzzles. The (hypothetical) approach is to first classify the squares according to types, where the types are based on the ways the squares are painted. Then, when a vacancy is to be filled that requires a particular kind of piece, the program examines various kinds of pieces that it has examples of and tests to see if the kind of piece found "ISA" piece of the type required. Describe how

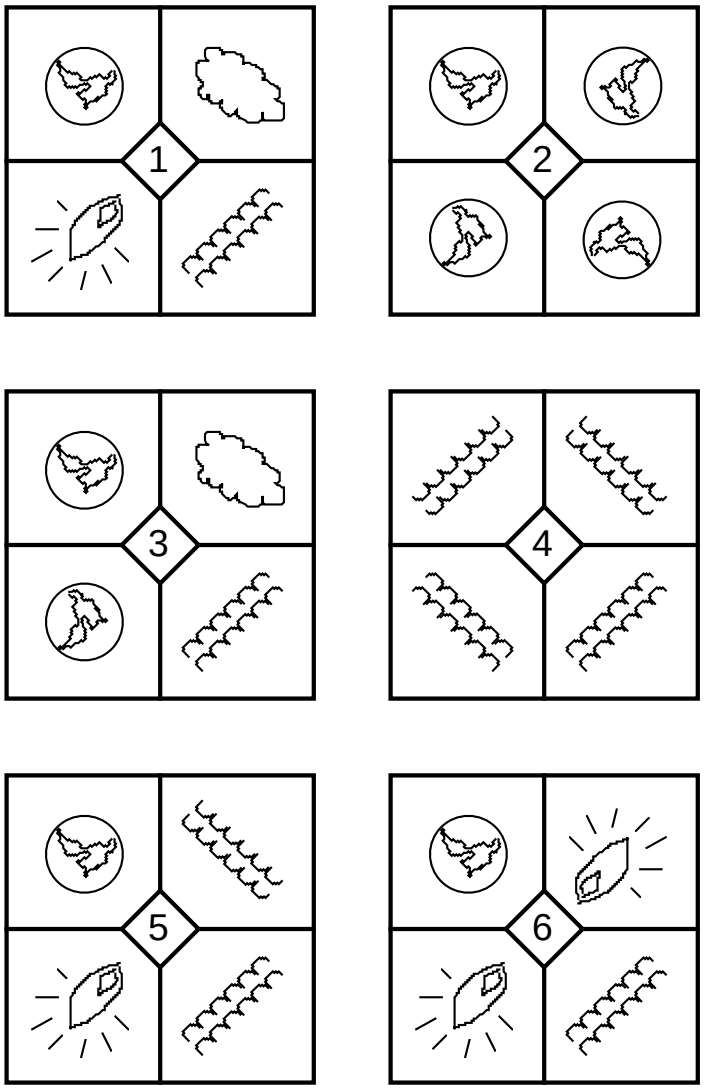

Figure 5.11: An instance of the equilibrium puzzle. In a solution, all pieces form a rectangle in which each interior junction has one each of the four elements, and each exterior junction has no repeated element.

types of painted squares could be defined so that an interesting inclusion hierarchy could be set up (without doing any programming).

9. A "straight-line dominoes" puzzle consists of a set of dominoes, and the object of the puzzle is to lay all the dominoes in a straight line so that the adjacent ends match. Discuss the difficulty of this kind of puzzle. How would a program to solve such puzzles be constructed?

10. The missionaries and cannibals puzzle goes as follows. Three missionaries and three cannibals are on one side of a river. They have a boat that can carry three people. At least one missionary is needed in the boat to row. They need to cross the river. The difficulty is that at no time is it safe to allow the cannibals to outnumber the missionaries on either side of the river or in the boat. If outnumbered, the missionaries would be eaten by the cannibals. How should they proceed to cross the river in a safe manner? Write a program that uses state-space search to solve the puzzle. The program should print out the solution with seven lines, the last two of which follow:

```
(CROSS RIVER WITH 1 MISSIONARIES AND 0 CANNIBALS)
(CROSS RIVER WITH 1 MISSIONARIES AND 1 CANNIBALS) (MADE IT!)
```

11. The four-peg version of the Towers of Hanoi puzzle is stated as follows: Four pegs, called A, B, C, and D, can each hold n rings at a time. There are n rings, $R_1, R_2, \ldots, R_n$, such that R_i is smaller than R_j whenever $i < j$. A legal placement of the rings on the pegs requires that (a) whenever any two rings appear on the same peg, the smaller one is above the larger one and (b) all n rings must be on pegs. The starting placement consists of all rings placed on peg A. The goal placement consists of all rings placed on peg D. Describe a representation for the states of the space to be searched. Describe a procedure that generates the successors of a state.

12. Using the data and function definitions in the text, compare the solutions found by the depth-first graph search method `dfgs` with those found by breadth-first search (`breadthfs`) for the problems of finding paths from Paris to Strasbourg, Strasbourg to Paris, Bordeaux to Lyon, and Lyon to Bordeaux.

    (a) What are the paths found?

    (b) What conclusions can you make?

    (c) Which procedure is more sensitive to the order in which the cities neighboring each city appear on the property list?

    (d) Describe alternative data for which the two procedures would perform equally efficiently.

13. Modify the function `dfgs`, and thus create a new function `quasiDFS`, by having the list `lst` concatenated onto the beginning of OPEN after culling members that are also present on either OPEN or CLOSED. The result of evaluating

    ```
    quasiDFS('Rennes', 'Avignon')
    ```

    should be

    ```
    ['Rennes', 'Caen', 'Calais', 'Nancy', 'Dijon', 'Lyon', 'Avignon']
    ```

    Find the result of evaluating:

    ```
    quasiDFS('Avignon', 'Rennes')
    ```

    If the adjacency information were reordered, would it be possible for `quasiDFS` to ever find the longest path from Rennes to Avignon? (Such a longest path has 13 arcs.) Describe the kinds of paths that can be found with this variation.

14. Adjust the parameters for `evolve` and try several runs with each set of values. What can you say about the influence of the values on the search?

15. Add a mechanism to the genetic search program to ensure that no two elements of the population are identical. Test its effect on the search process.

16. Add a mechanism to generate a new kind of mutation: It should randomly select a position `where` and then rotate the cities in the left part (the part to the left of `where`) of the path. Test its effect on the search process.

17. Write a program to explore the subject of "genetic art" as described in [Sims 1991] where each member of the population is a mathematical expression expression that generates an image and the user interactively performs "quasi-Darwinian" selection.

18. The (n, k) version of "Last One Loses" starts with a stack of pennies, n high. Two players, "Plus" and "Minus," alternate moves. Minus makes the first move. In each move, a player may remove up to k pennies from the stack but must remove at least 1. The player stuck with the last move is the loser. Let the game configuration be represented as $W(N)\sigma$, where N denotes the current number of pennies left, and σ is either "+" or "−" and indicates whose move it is. Note that $W(0)+$ and $W(0)-$ can never be reached, since $W(1)+$ and $W(1)-$ are terminal configurations.

    (a) Draw the AND/OR graph for the case $(n, k) = (9, 3)$.

(b) Superimpose the solution graph for the same case (by hatching selected arcs of the AND/OR graph) that proves Plus can always force a win.

(c) For which (n, k) pairs is Last One Loses a determined game? Who wins in each case? By what strategy?

19. The game tree in Figure 5.12 illustrates the possible moves, to a depth of 4, that can be made from the current position (at the root) in a hypothetical game between a computer and a human. The evaluation function is such that the computer seeks to maximize the score while the human seeks to minimize it. The computer has 5 seconds to make its move and 4 of these have been allocated to evaluating board positions. The order in which board positions are evaluated is determined by the root's being "searched." A node that is at ply 0, 1, 2, or 3 is *searched* by

- generating its children
- statically evaluating the children
- ordering its children by static value, in either ascending or descending order, so as to maximize the probability of alpha or beta cutoffs during the searches of successive children
- searching the children if they are not in ply 4, and
- backing up the minimum or maximum value from the children, where the value from each child is the backed-up value (if the child is not in ply 4) or the static value (if the child is in ply 4)

The computer requires 1/7 second to statically evaluate a node. Other times are negligible (move generation, backing up values, etc.). The computer chooses the move (out of those whose backed-up values are complete) having the highest backed-up value.

(a) Give the order in which nodes will be statically evaluated (indicate the ith node by putting the integer i in the circle for that node). Hint: the first eight nodes have been done for you. Be sure to skip the nodes that alpha-beta pruning would determine as irrelevant. Indicate where cutoffs occur.

(b) Determine the backed-up values for the relevant nodes. (Fill in the squares.) Node Q has been done for you.

(c) Keeping in mind that it takes 1/7 second per static evaluation, what will be the computer's move?

(d) Now assume that it takes 1/8 second per static evaluation. What will be the computer's move?

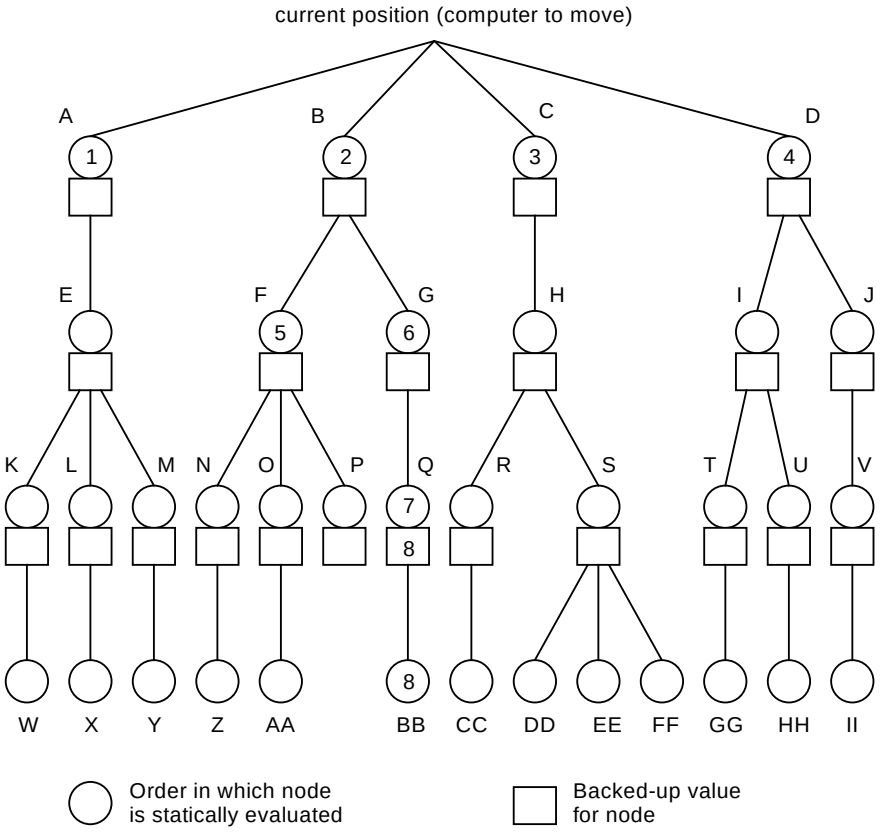

Figure 5.12: Hypothetical game tree. Static values for nodes: A 4, B 15, C 13, D 10, E 20, F 9, G 8, H 10, I 10, J 8, K 5, L 20, M 3, N 7, O 6, P 0, Q 9, R 12, S 10, T 15, U 10, V 9, W 7, X 22, Y 2, Z 7, AA 5, BB 8, CC 15, DD 12, EE 13, FF 13, GG 20, HH 22, II 18.

20. Write a Python program that plays "tic-tac-toe" according to the scheme described on page 200.

    - You should be able to easily set the maximum depth of the program's search to any given ply.
    - Include a function PRINT-BOARD that shows a given board position in a neat format.
    - Illustrate your program's behavior for eight games as follows: two games at each of the four following maximum search depths: 1, 2, 4, and 6. In each pair, the human should play differently. In each game, show the board position after each of the actual moves chosen by your program. You may have the program play either O's or X's, but indicate which side the program is playing.
    - After each board position printed, print out the number of times the evaluation function was computed to determine the move just made.
    - At the end of each game, print out the number of times the evaluation function was applied to board positions during that game.
    - Describe the trade-off you observe between computation time (as measured by the number of computations of the evaluation function) and the skill of the program.

21. (Term Project) Write a program to play "Baroque Chess." This game is played with conventional chess pieces and set up as usual, except that the King's side rook for each player is turned upside-down. However, the names of the pieces are changed, as are the rules of the game. The Baroque Chess names of the pieces are given below following their common names.

Pawn	=	Squeezer
Knight	=	Leaper
Bishop	=	Imitator
(right-side-up) Rook	=	Freezer
(upside-down) Rook	=	Coordinator
Queen	=	Step-back
King	=	King

    Most of the pieces may move like the Queens of ordinary chess. Exceptions are the Squeezers, Leapers (when capturing), and the Kings. However, many pieces have special requirements for making captures. The particular characteristics of each kind of piece are now given.

    - Squeezer—moves like a rook of ordinary chess, vertically or horizontally any number of squares. In order to capture an opponent's piece, the Squeezer must be moved so as to "sandwich" the piece between the Squeezer and another Squeezer. Two Squeezers sandwich

an opposing piece by being on either side of it, either horizontally or vertically.

- Freezer—moves like a Queen. It does not capture other pieces. However, when it is adjacent (in any of the eight neighboring squares) to an opponent's piece, the opponent may not move that piece.
- Coordinator—moves like a Queen. The Coordinator's row and the same player's King's column determine the location of a square on the board. If the coordinator is moved so as to make this square one where an opponent's piece stands, the piece is captured.
- Leaper—moves like a Queen, except that when capturing, it must complete its move by jumping over the piece it captures to the next square in the same line; that square must be vacant to permit the capture.
- Step-back—moves (and looks) like a Queen. However, in order to capture a piece, it must begin its move in a position adjacent to the piece (i.e., in any of the eight neighboring squares), and then it must move exactly one square in the direction away from the piece.
- King—moves (and looks) like a normal chess King, and it captures like a normal chess King (however, there is no "castling" move in Baroque Chess). The game is finished when a King is captured (there is no checkmating or need to say "check" in Baroque Chess).
- Imitator—normally moves like a Queen. However, in order to capture a piece, an Imitator must do as the captured piece would do to capture. In addition, if an Imitator is adjacent to the opponent's Freezer, the Imitator freezes the Freezer, and then neither piece may be moved until one of the two is captured.

Making the game even more interesting is a rule that makes all the captures implied by a move effective. For example, a Squeezer may move to simultaneously sandwich two opposing pieces and capture both of them. Another example would be a situation where an Imitator, in one move, steps back from a Step-back (capturing it) and in its new position sandwiches a Squeezer and captures it.