PART 3
Knowledge Organization and Manipulation

9
Search and Control Strategies

In the next three chapters we examine the organization and manipulation of knowledge. This chapter is concerned with search, an operation required in almost all AI programs. Chapter 10 covers the comparison or matching of data structures and in particular pattern matching, while Chapter 11 is concerned with the organization of knowledge in memory.

Search is one of the operational tasks that characterize AI programs best. Almost every AI program depends on a search procedure to perform its prescribed functions. Problems are typically defined in terms of states, and solutions correspond to goal states. Solving a problem then amounts to searching through the different states until one or more of the goal states are found. In this chapter we investigate search techniques that will be referred to often in subsequent chapters.

9.1 INTRODUCTION

Consider the process of playing a game such as chess. Each board configuration can be thought of as representing a different state of the game. A change of state occurs when one of the players moves a piece. A goal state is any of the possible board configurations corresponding to a checkmate.

It has been estimated that the game of chess has more than 10^{120} possible

states. (To see this, just note that there are about 20 alternative moves for each board configuration and more than 100 different configurations. Thus, there are more than $20^{100} = 10^{100} * 2^{100} > 10^{120}$). This is another example of the combinatorial explosion problem. The number of states grows exponentially with the number of basic elements. Winning a game amounts to finding a sequence of states through this maze of possible states that leads to one of the goal states.

An "intelligent" chess playing program certainly would not play the game by exploring all possible moves (it would never finish in our lifetime nor in your distant descendent's lifetimes). Like a human, the program must eliminate many questionable states when playing. But, even with the elimination of numerous states, there is still much searching to be done since finding good moves at each state of the game often requires looking ahead a few moves and evaluating the consequences.

This type of problem is not limited to games. Search is ubiquitous in AI. For every interesting problem there are numerous alternatives to consider. When attempting to understand a natural language, a program must search to find matching words that are known (a dictionary), sentence constructions, and matching contexts. In vision perception, program searches must be performed to find model patterns that match input scenes. In theorem proving, clauses must be found by searching axioms and assertions which resolve together to give the empty clause. This requires a search of literals which unify and then a search to find resolvable clauses. In planning problems, a number of potential alternatives must be examined before a good workable plan can be formulated. As in learning, many potential hypotheses must be considered before a good one is chosen.

9.2 PRELIMINARY CONCEPTS

Problems can be characterized as a space consisting of a set of states (not necessarily finite) and a set of operators that map from one state to other states. Three types of states may be distinguished: one or more initial states, a number of intermediate states, and one or more goal states. A solution to a problem is a sequence of operators that map an initial state to a goal state. A "best" or good solution is one that requires the fewest operations or the least cost to map from an initial state to a goal state. The performance of a particular solution method is judged by the amount of time and memory space required to complete the mapping. Thus, a solution based on some algorithm A_1 is considered better than one using algorithm A_2 if the time and space complexity of A_1 is less than that of A_2.

Time and Space Complexity

Time and space complexities of algorithms may be defined in terms of their best, their average, or their worst-case performance in completing some task. In evaluating different search strategies, we follow the usual convention of considering worst-

case performances and look for ways to improve on them. For this, we need the O (for order) notation.

Let f and g be functions of n, where algorithm A has size n. The size can be the number of problem states, the number of input characters which specify the problem or some similar number. Let $f(n)$ denote the time (or space) required to solve a given problem using algorithm A. We say "f is big O of g" written $f = O(g)$, if and only if there exists a constant $c > 0$ and an integer n_0, such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Stated more simply, algorithm A solves a problem in at most $cg(n)$ units or steps for all but a finite number of steps. Based on this definition, we say an algorithm is of linear time if it is $O(n)$. It is of quadratic time if it is $O(n^2)$, and of exponential time if it is $O(2^{kn})$ for some constant k (or if it is $O(b^{kn})$ for any real number $b > 1$).

For example, if a knowledge base has ten assertions (clauses), with an average of five literals per clause, and a resolution proof is being performed with no particular strategy, a worst-case proof may require as many as 1125 comparisons ($5^2 \times 10(9)/2$) for a single resolution and several times this number for a complete proof.

Graph and Tree Representations

It is customary to represent a search space as a diagram of a directed graph or a tree. Each node or vertex in the graph corresponds to a problem state, and arcs between nodes correspond to transformations or mappings between the states. The immediate successors of a node are referred to as children, siblings, or offspring, and predecessor nodes are ancestors. An immediate ancestor to a node is a parent.

A tree is a graph in which each node has at most one parent. One node, the root or starting node, has no parent. Leaf or terminal nodes are nodes without children. The number of successors emanating from a node is called the branching degree of that node (denoted as b). A path is a sequence of nodes $n_1, \ldots, n_k$, where each n_i is a successor of n_{i-1} for $i = 1, \ldots, k$.

It is always possible to convert a directed graph into a tree with multiple labeled nodes. This can be done by opening up all but one of the several alternate paths connecting two nodes and creating duplicate copies of the end node, one for each different path from the parent. We will find it more convenient, however, to use both types of representations in the following discussion.

An And-Or graph or tree is a special type of representation for problems which can be reduced to a set of subproblems, all of which must be solved. The requirement for the solution of all subproblems is depicted as an And node, a node with all arcs emanating from it connected by a curved line. Or nodes have no line connecting its arcs to signify that any emanating path may be taken for a solution. For example, if a robot is given the task of painting a table, it may complete the task by scraping, sanding, and painting the table, or it may choose the simpler solution and send it to a paintshop (Figure 9.1).

In what follows, we assume simple Or graphs or trees as the problem space

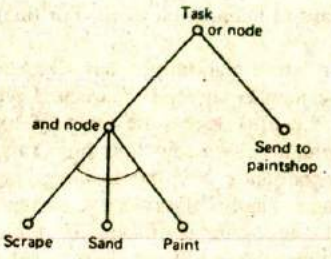


Figure 9.1 Example of an and-or graph.

representation unless noted otherwise. And-Or graph searches are covered in Section 9.6.

Graph and Search Trees

Search can be characterized as finding a path through a graph or tree structure. This requires moving from node to node after successively expanding and generating connected nodes. Node generation is accomplished by computing the identification or representation code of children nodes from a parent node. Once this is done, a child is said to be *generated* and the parent is said to be *explored*. The process of generating all of the children of a parent is also known as *expanding* the node. A search procedure is a strategy for selecting the order in which nodes are generated and a given path selected.

Search problems may be classified by the information used to carry out a given strategy. In *blind or uninformed search*, no preference is given to the order of successor node generation and selection. The path selected is blindly or mechanically followed. No information is used to determine the preference of one child over another.

In *informed or directed search*, some information about the problem space is used to compute a preference among the children for exploration and expansion. Before proceeding with a comparison of strategies, we consider next some typical search problems.

9.3 EXAMPLES OF SEARCH PROBLEMS

In this section we describe three typical problems which illustate the concepts defined above and which are used in subsequent sections to portray different search techniques. The problems considered are the often-used examples, the eight puzzle and the traveling salesman problem.

The Eight Puzzle

The eight puzzle consists of a 3-by-3 square frame which holds eight movable square tiles which are numbered from 1 to 8. One square is empty, permitting tiles

Figure 9.2 The eight puzzle game.

to be shifted (Figure 9.2). The objective of the puzzle is to find a sequence of tile movements that leads from a starting configuration to a goal configuration such as that shown in Figure 9.2.

The states of the eight puzzle are the different permutations of the tiles within the frame. The operations are the permissible moves (one may consider the empty space as being moveable rather than the tiles): up, down, left, and right. An optimal or good solution is one that maps an initial arrangement of tiles to the goal configuration with the smallest number of moves.

The search space for the eight puzzle problem may be depicted as the tree shown in Figure 9.3.

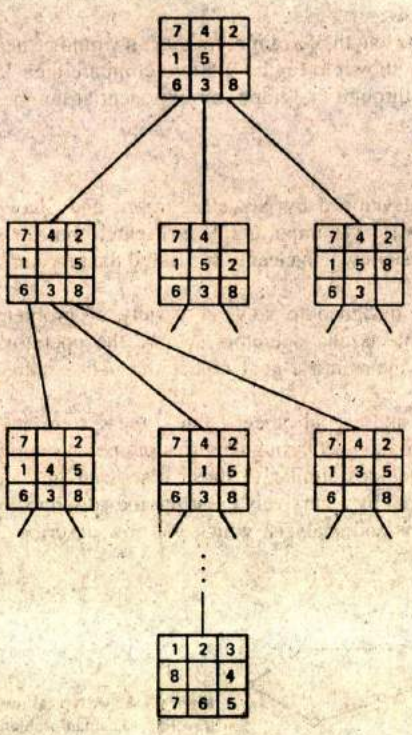


Figure 9.3 A tree diagram for the eight puzzle.

In the figure, the nodes are depicted as puzzle configurations. The root node represents a randomly chosen starting configuration, and its successor nodes correspond to the three single tile movements that are possible from the root. A path is a sequence of nodes starting from the root and progressing downward to the goal node.

Traveling Salesman Problem

The traveling salesman problem involves n cities with paths connecting the cities. A tour is any path which begins with some starting city, visits each of the other cities exactly once, and returns to the starting city. A typical tour is depicted in Figure 9.4.

The objective of a traveling salesman problem is to find a minimal distance tour. To explore all such tours requires an exponential amount of time. For example, a minimal solution with only 10 cities is tractable (3,628,000 tours). One with 20 or more cities is not, since a worst-case search requires on the order of 20! (about 23×10^{17}) tours. The state space for the problem can also be represented as a graph as depicted in Figure 9.5.

Without knowing in advance the length of a minimum tour, it would be necessary to traverse each of the distinct paths shown in Figure 9.5 and compare their lengths. This requires some $O(n!)$ traverses through the graph, an exponential number.

General Problem Solver

The General Problem Solver was developed by Newell, Simon, and Shaw (Ernst and Newell, 1969) in the late 1950s. It was important as a research tool for several reasons and notable as the first AI system which cleanly separated the task knowledge from the problem solving part.

General Problem Solver was designed to solve a variety of problems that could be formulated as a set of objects and operators, where the operators were applied to the objects to transform them into a goal object through a sequence of applications.

Given an initial object (state) and a goal object (state), the system attempted to transform the initial object to the goal object through a series of operator application transformations. It used a set of methods similar to those discussed in Chapter 8 for each goal type, to achieve that goal by recursively creating and solving subgoals. The basic method is known as means-end analysis, which we now describe.

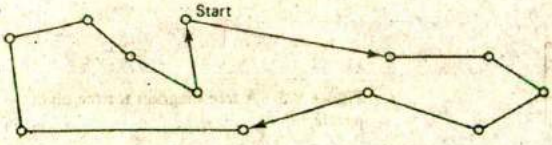


Figure 9.4 A typical tour for the traveling salesman problem.

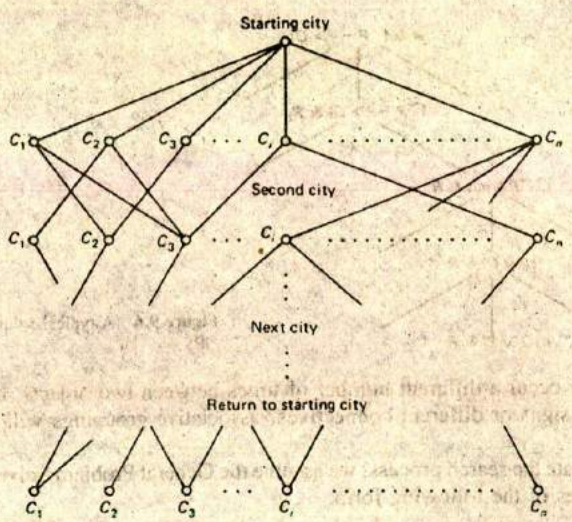


Figure 9.5 State space representation for the TSP.

Means-end analysis. The problem space of means-end analysis has an initial state (object) and one or more goal states (objects), a set of operators O_k with given preconditions for their application, and a difference function that computes the difference between two states S_i and S_j. A problem is solved using means-end analysis by:

1. Comparing the current state S_i to a goal state S_g and computing the difference D_{ig}.
2. An operator O_k is then selected to reduce the difference D_{ig}.
3. The operator O_k is applied if possible. If not, the current state is saved, a subgoal is created and means-end analysis is applied recursively to reduce the subgoal.
4. If the subgoal is solved, the saved state is restored and work is resumed on the original problem.

In carrying out these methods, the General Problem Solver may transform some S_i into an intermediate state S_k, to reduce the difference D_{ij} between states S_i and S_j, then apply another operator O_l to the S_k, and so on until the state S_j is obtained. Differences that may occur between objects will, of course, depend on the task domain.

As an example, in proving theorems in propositional logic, some common differences that occur are a variable may appear in one object and not in the other,

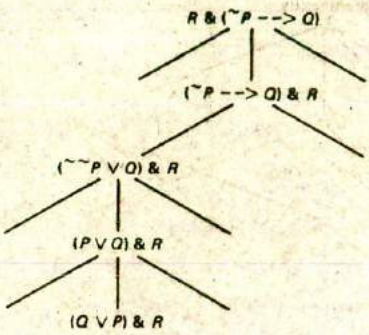


Figure 9.6 A typical solution tree for GPS.

a variable may occur a different number of times between two objects, objects will have different signs or different connectives, associative groupings will differ, and so on.

To illustrate the search process, we assume the General Problem Solver operators are rewrite rules of the following form:

R1: $(A \vee B) \rightarrow (B \vee A)$
R2: $(A \& B) \rightarrow (B \& A)$
R3: $(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$
R4: $(A \rightarrow B) \rightarrow (\neg A \vee B)$

As a simple example, we suppose General Problem Solver is given the initial propositional logic object $L_i = (R \& (\neg P \rightarrow Q))$ and goal object $L_g = ((Q \vee P) \& R)$. To determine L_g from L_i requires a few simple transformations. The system first determines the difference between the two expressions and then systematically reduces these differences until L_g is obtained from L_i or failure occurs. For example, a comparison of L_i and L_g reveals the difference that R is on the left in L_i but on the right in L_g. This causes a subgoal to be set up to reduce this difference. The subgoal, in turn, calls for an application of the reduction method, namely to rewrite L_i in the equivalent form $L'_i = ((\neg P \rightarrow Q) \& R)$. The rest of the solution process follows the path indicated in the tree of Figure 9.6.

9.4 UNINFORMED OR BLIND SEARCH

As noted earlier, search problems can be classified by the amount of information that is available to the search process. Such information might relate to the problem space as a whole or to only some states. It may be available a priori or only after a node has been expanded.

In a worst case situation the only information available will be the ability to distinguish goal from nongoal nodes. When no further information is known a priori, a search program must perform a blind or uninformed search. A blind or uninformed search algorithm is one that uses no information other than the initial state, the search operators, and a test for a solution. A blind search should proceed in a systematic way by exploring nodes in some predetermined order or simply by selecting nodes at random. We consider only systematic search procedures in this section.

Search programs may be required to return only a solution value when a goal is found or to record and return the solution path as well. To simplify the descriptions that follow, we assume that only the goal value is returned. To also return the path requires making a list of nodes on the path or setting back-pointers to ancestor nodes along the path.

Breadth-First Search

Breadth-first searches are performed by exploring all nodes at a given depth before proceeding to the next level. This means that all immediate children of nodes are explored before any of the children's children are considered. Breadth first tree search is illustrated in Figure 9.7. It has the obvious advantage of always finding a minimal path length solution when one exists. However, a great many nodes may need to be explored before a solution is found, especially if the tree is very full.

An algorithm for the breadth-first search is quite simple. It uses a queue structure to hold all generated but still unexplored nodes. The order in which nodes are placed on the queue for removal and exploration determines the type of search. The breadth-first algorithm proceeds as follows.

BREADTH-FIRST SEARCH

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,

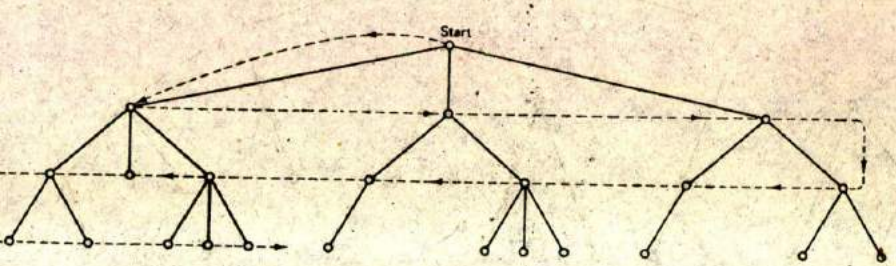


Figure 9.7 Breadth-first search of a tree.

4. Remove and expand the first element from the queue and place all the children at the end of the queue in any order.
5. Return to step 2.

The time complexity of the breadth-first search is $O(b^d)$. This can be seen by noting that all nodes up to the goal depth d are generated. Therefore, the number generated is $b + b^2 + \ldots + b^d$ which is $O(b^d)$. The space complexity is also $O(b^d)$ since all nodes at a given depth must be stored in order to generate the nodes at the next depth, that is, b^{d-1} nodes must be stored at depth $d - 1$ to generate nodes at depth d, which gives space complexity of $O(b^d)$. The use of both exponential time and space is one of the main drawbacks of the breadth-first search.

Depth-First Search

Depth-first searches are performed by diving downward into a tree as quickly as possible. It does this by always generating a child node from the most recently expanded node, then generating that child's children, and so on until a goal is found or some cutoff depth point d is reached. If a goal is not found when a leaf node is reached or at the cutoff point, the program backtracks to the most recently expanded node and generates another of its children. This process continues until a goal is found or failure occurs.

An example of a depth-first search is illustrated in Figure 9.8.

An algorithm for the depth-first search is the same as that for breadth-first except in the ordering of the nodes placed on the queue. Depth-first places the newly generated children at the head of the queue so that they will be chosen first. The search proceeds as follows.

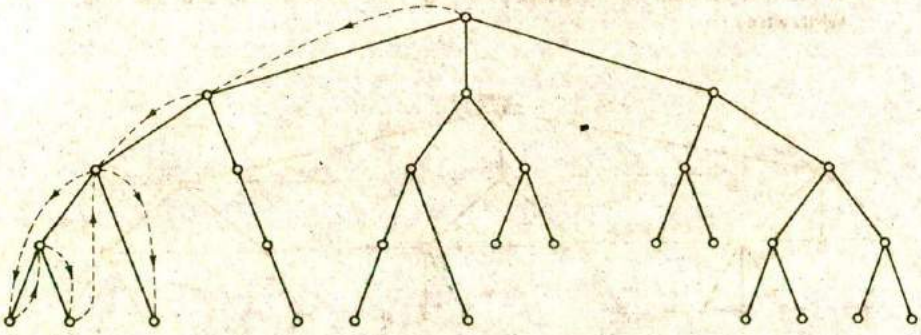


Figure 9.8 Depth-first search of a tree.

DEPTH-FIRST SEARCH

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element, and place the children at the front of the queue (in any order).
5. Return to step 2.

The depth-first search is preferred over the breadth-first when the search tree is known to have a plentiful number of goals. Otherwise, depth-first may never find a solution. The depth cutoff also introduces some problems. If it is set too shallow, goals may be missed; if set too deep, extra computation may be performed.

The time complexity of the depth-first tree search is the same as that for breadth-first, $O(b^d)$. It is less demanding in space requirements, however, since only the path from the starting node to the current node needs to be stored. Therefore, if the depth cutoff is d, the space complexity is just $O(d)$.

Depth-First Iterative Deepening Search

Depth-first iterative deepening searches are performed as a form of repetitive depth first search moving to a successively deeper depth with each iteration. It begins by performing a depth-first search to a depth of one. It then discards all nodes generated and starts over doing a search to a depth of two. If no goal has been found, it discards all nodes generated and does a depth-first search to a depth of three. This process continues until a goal node is found or some maximum depth is reached.

Since the depth-first iterative deepening search expands all nodes at a given depth before expanding nodes at a greater depth, it is guaranteed to find a shortest-path solution. The main disadvantage of this method is that it performs wasted computations before reaching a goal depth. Even so, it has been shown to be asymptotically optimal over depth and breadth first search in terms of time and space complexity (Korf, 1985). That is, depth- and breadth-first searches take at least as much time and memory as depth-first iterative deepening searches for increasingly large searches. The time and space complexities of this search are $O(b^d)$ and $O(d)$ respectively.

This search algorithm works basically the same as the depth first search algorithm given above for a single iteration. However, it terminates the search at depth d on each iteration if no goal has been found, removes all nodes from the queue, increments d by one, and initiates the search again.

Bidirectional Search

When a problem has a single goal state that is given explicitly, and all node generation operators have inverses, bidirectional search can be used. (This is the case with

the eight puzzle described above, for example). Bidirectional search is performed by searching forward from the initial node and backward from the goal node simultaneously. To do so, the program must store the nodes generated on both search frontiers until a common node is found. With some modifications, all three of the blind search methods described above may be used to perform bidirectional search.

For example, to perform bidirectional depth-first iterative deepening search to a depth of k, the search is made from one direction and the nodes at depth k are stored. At the same time, a search to a depth of k and $k + 1$ is made from the other direction and all nodes generated are matched against the nodes stored from the other side. These nodes need not be stored, but a search of the two depths is needed to account for odd-length paths. This process is repeated for lengths $k = 0$ to $d/2$ from both directions.

The time and space complexities for bidirectional depth-first iterative deepening search are both $O(b^{d/2})$ when the node matching is done in constant time per node.

Since the number of nodes to be searched using the blind search methods described above increase as b^d with depth d, such problems become intractable for large depths. It, therefore, behooves us to consider alternative methods. Such methods depend on some knowledge to limit the number of problem states visited. We turn to these methods now in the next section.

9.5 INFORMED SEARCH

When more information than the initial state, the operators, and the goal test is available, the size of the search space can usually be constrained. When this is the case, the better the information available, the more efficient the search process will be. Such methods are known as informed search methods. They often depend on the use of heuristic information. In this section, we examine search strategies based on the use of some problem domain information, and in particular, on the use of heuristic search functions.

Heuristic Information

Information about the problem (the nature of the states, the cost of transforming from one state to another, the promise of taking a certain path, and the characteristics of the goals) can sometimes be used to help guide the search more efficiently. This information can often be expressed in the form of a heuristic evaluation function $f(n,g)$, a function of the nodes n and/or the goals g.

Recall that a heuristic is a rule of thumb or judgmental technique that leads to a solution some of the time but provides no guarantee of success. It may in fact end in failure. Heuristics play an important role in search strategies because of the exponential nature of most problems. They help to reduce the number of alternatives from an exponential number to a polynomial number and, thereby, obtain a solution

in a tolerable amount of time. When exhaustive search is impractical, it is necessary to compromise for a constrained search which eliminates many paths but offers the promise of success some of the time. Here, success may be considered to be finding an optimal solution a fair proportion of the time or just finding good solutions much of the time. In this regard, any policy which uses as little search effort as possible to find any qualified goal has been called a *satisficing policy*.

Consider for example, the traveling salesman problem described above. A simple heuristic for choosing the next city at any point in a tour is one which picks the nearest unvisited neighbor. This policy gives no guarantee of an optimal solution, but its solutions are often good, and the time required is only $O(n^2)$. Likewise, for the eight puzzle, using a heuristic function, which selects moves that produce the smallest number of tiles out of place from the goal configuration, can result in a worthwhile time saving. In solving a problem in propositional logic, such as proving a theorem in the General Problem Solver, the time complexity can often be reduced from exponential to polynomial time through the application of a simple heuristic strategy. In General Problem Solver this is accomplished by first planning a solution by breaking the main problem down into several subproblems of lesser complexity. This often has the effect of reducing the overall complexity by several orders of magnitude.

Hill Climbing Methods

Search methods based on hill climbing get their names from the way the nodes are selected for expansion. At each point in the search path, a successor node that appears to lead most quickly to the top of the hill (the goal) is selected for exploration. This method requires that some information be available with which to evaluate and order the most promising choices.

Hill climbing is like depth-first searching where the most promising child is selected for expansion. When the children have been generated, alternative choices are evaluated using some type of heuristic function. The path that appears most promising is then chosen and no further reference to the parent or other children is retained. This process continues from node-to-node with previously expanded nodes being discarded. A typical path is illustrated in Figure 9.9 where the numbers by a node correspond to the computed estimates of the goal distance for alternative paths.

Hill climbing can produce substantial savings over blind searches when an informative, reliable function is available to guide the search to a global goal. It suffers from some serious drawbacks when this is not the case. Potential problem types named after certain terrestrial anomalies are the foothill, ridge, and plateau traps.

The foothill trap results when local maxima or peaks are found. In this case the children all have less promising goal distances than the parent node. The search is essentially trapped at the local node with no indication of goal direction. The only way to remedy this problem is to try moving in some arbitrary direction a few generations in the hope that the real goal direction will become evident, backtrack-

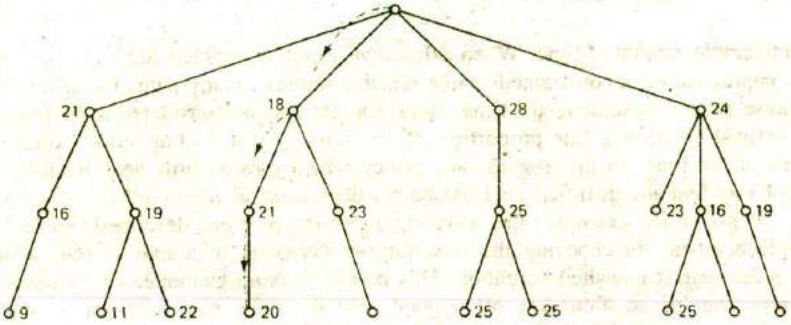


Figure 9.9 Search with hill climbing.

ing to an ancestor node and trying a secondary path choice, or altering the computation procedure to expand ahead a few generations each time before choosing a path.

A second potential problem occurs when several adjoining nodes have higher values than surrounding nodes. This is the equivalent of a ridge. It too is a form of local trap and the only remedy is to try to escape as in the foothill case above.

Finally, the search may encounter a plateau type of structure, that is, an area in which all neighboring nodes have the same values. Once again, one of the methods noted above must be tried to escape the trap.

The problems encountered with hill climbing can be avoided using a best-first search approach.

Best-First Search

Best-first search also depends on the use of a heuristic to select most promising paths to the goal node. Unlike hill climbing, however, this algorithm retains all estimates computed for previously generated nodes and makes its selection based on the best among them all. Thus, at any point in the search process, best-first moves forward from the most promising of all the nodes generated so far. In so doing, it avoids the potential traps encountered in hill climbing. The best-first process is illustrated in Figure 9.10 where numbers by the nodes may be regarded as estimates of the distance or cost to reach the goal node.

The algorithm we give for best first search differs from the previous blind search algorithms only in the way the nodes are saved and ordered on the queue. The algorithm reads as follows.

BEST-FIRST SEARCH

1. Place the starting node *s* on the queue.
2. If the queue is empty, return failure and stop.

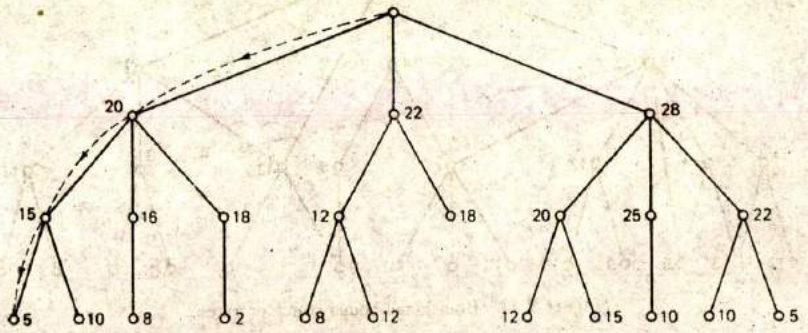


Figure 9.10 Best-first search of a tree.

3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove the first element from the queue, expand it and compute the estimated goal distances for each child. Place the children on the queue (at either end) and arrange all queue elements in ascending order corresponding to goal distance from the front of the queue.
5. Return to step 2.

Best-first searches will always find good paths to a goal, even when local anomalies are encountered. All that is required is that a good measure of goal distance be used.

Branch-and-Bound Search

The branch-and-bound search strategy applies to problems having a graph search space where more than one alternative path may exist between two nodes. This strategy saves all path lengths (or costs) from a node to all generated nodes and chooses the shortest path for further expansion. It then compares the new path lengths with all old ones and again chooses the shortest path for expansion. In this way, any path to a goal node is certain to be a minimal length path. This process is illustrated in Figure 9.11.

An algorithm for the branch-and-bound strategy which uses a queue data structure to hold partial paths developed during the search is as follows.

BRANCH-AND-BOUND SEARCH

1. Place the start node of zero path length on the queue.
2. Until the queue is empty or a goal node has been found: (a) determine if the first path in the queue contains a goal node, (b) if the first path contains a goal node exit with success, (c) if the first path does not contain a goal node,

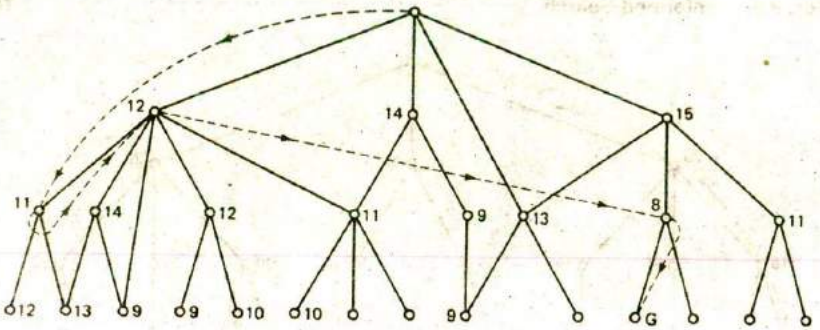


Figure 9.11 Branch-and-bound search of a tree.

remove the path from the queue and form new paths by extending the removed path by one step, (d) compute the cost of the new paths and add them to the queue, (e) sort the paths on the queue with lowest-cost paths in front.

3. Otherwise, exit with failure.

Always extending the lowest-cost path in branch-and-bound search insures that a lowest-cost path will be found if one exists. Of course, this is at the expense of computing and remembering all competing paths. We next look at a special case of branch-and-bound search which estimates the total cost to a goal node, and selects the least cost path at each stage in the search.

Optimal Search and A*

The previous heuristic methods offer good strategies but fail to describe how the shortest distance to a goal should be estimated. The A* algorithm is a specialization of best-first search. It provides general guidelines with which to estimate goal distances for general search graphs.

At each node along a path to the goal, the A* algorithm generates all successor nodes and computes an estimate of the distance (cost) from the start node to a goal node through each of the successors. It then chooses the successor with the shortest estimated distance for expansion. The successors for this node are then generated, their distances estimated, and the process continues until a goal is found or the search ends in failure.

The form of the heuristic estimation function for A* is

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

where the two components $g^*(n)$ and $h^*(n)$ are estimates of the cost (or distance) from the start node to node n and the cost from node n to a goal node, respectively. The asterisks are used to designate estimates of the corresponding true values $f(n)$

$= g(n) + h(n)$. For state space tree problems $g^*(n) = g(n)$ since there is only one path and the distance $g^*(n)$ will be known to be the true minimum from the start to the current node n. This is not true in general for graphs, since alternate paths from the start node to n may exist.

For this type of problem, it is convenient to maintain two lists of node types designated as open and closed. Nodes on the open list are nodes that have been generated but not yet expanded while nodes on the closed list are nodes that have been expanded and whose children are, therefore, available to the search program. The A* algorithm proceeds as follows.

A* SEARCH

1. Place the starting node s on open.
2. If open is empty, stop and return failure.
3. Remove from open the node n that has the smallest value of $f^*(n)$. If the node is a goal node, return success and stop. Otherwise,
4. Expand n, generating all of its successors n' and place n on closed. For every successor n', if n' is not already on open or closed attach a back-pointer to n, compute $f^*(n')$ and place it on open.
5. Each n' that is already on open or closed should be attached to back-pointers which reflect the lowest $g^*(n')$ path. If n' was on closed and its pointer was changed, remove it and place it on open.
6. Return to step 2.

Next, we consider some desirable properties of heuristic search algorithms. They are summarized in the following definitions.

Admissibility condition. Algorithm A is *admissible* if it is guaranteed to return an optimal solution when one exists.

Completeness condition. Algorithm A is *complete* if it always terminates with a solution when one exists.

Dominance property. Let A_1 and A_2 be admissible algorithms with heuristic estimation functions h^*_1 and h^*_2, respectively. A_1 is said to be more *informed* than A_2 whenever $h^*_1(n) > h^*_2(n)$ for all n. A_1 is also said to dominate A_2.

Optimality Property. Algorithm A is *optimal* over a class of algorithms if A dominates all members of the class.

The admissibility condition for an algorithm has led to a corresponding definition for a heuristic function h^*; h^* is said to be admissible if $h^* \leq h$ for all n. It can be shown that if A_1 and A_2 are admissible, and A_1 is more informed than A_2, then A_1 never expands a node not expanded by A_2. In general then, it is desirable to

find admissible heuristic functions that approximate h as closely as possible. This will insure that few if any nodes off the optimal path are expanded. Of course, if $h^* = h$ only nodes on the optimal path will be expanded. The cost of computing such a function should also be taken into account, however. It may not be cost effective if the computation cost is too high.

It has been shown that the A* algorithm is both complete and admissible. Thus, A* will always find an optimal path if one exists. The efficiency of an A* algorithm depends on how closely h^* approximates h and the cost of the computing f^*.

Iterative Deepening A*

By combining a heuristic evaluation function with a modified version of the iterative deepening search method presented earlier, we obtain iterative deepening A* or IDA*.

IDA* performs a depth search at each iteration and eliminates or trims all branches whose estimated cost ($g^* + h^*$) exceeds a given threshold $T(i)$ where $i = 0,1,2 \ldots$ is the iteration number. The initial threshold $T(0)$ is the estimated cost of the initial state. After that, the threshold increases with each iteration. The value of T on iteration $i + 1$ is taken as the minimum of the costs which exceed T on iteration i.

Like A*, it can be shown that IDA* always finds a cheapest path if h^* is admissible. Furthermore, IDA* expands the same number of nodes as A* (asymptotically).

9.6 SEARCHING AND-OR GRAPHS

The depth-first and breadth-first strategies given earlier for Or trees and graphs can easily be adapted for And-Or trees. The main difference lies in the way termination conditions are determined, since all goals following an And node must be realized, whereas a single goal node following an Or node will do. Consequently, we describe a more general optimal strategy that subsumes these types, the AO* (O for ordered) algorithm.

As in the case of the A* algorithm, we use the open list to hold nodes that have been generated but not expanded and the closed list to hold nodes that have been expanded (successor nodes that are available). The algorithm is a variation of the original given by Nilsson (1971). It requires that nodes traversed in the tree be labeled as solved or unsolved in the solution process to account for And node solutions which require solutions to all successor nodes. A solution is found when the start node is labeled as solved.

THE AO* ALGORITHM

1. Place the start node s on open.
2. Using the search tree constructed thus far, compute the most promising solution tree T_0.

3. Select a node n that is both on open and a part of T_0. Remove n from open and place it on closed.
4. If n is a terminal goal node, label n as solved. If the solution of n results in any of n's ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where T_0 is the solution tree. Remove from open all nodes with a solved ancestor.
5. If n is not a solvable node (operators cannot be applied), label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. If any of n's ancestors become unsolvable because n is, label them unsolvable as well. Remove from open all nodes with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successors. For each such successor node that contains more than one subproblem, generate their successors to give individual subproblems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h^* for each newly generated node and place all such nodes that do not yet have descendents on open. Next, recompute the values of h^* at n and each ancestor of n.
7. Return to step 2.

It can be shown that AO* will always find a minimum-cost solution tree if one exists, provided only that $h^*(n) \leq h(n)$, and all arc costs are positive. Like A*, the efficiency depends on how closely h^* approximates h.

9.7 SUMMARY

Search is characteristic of almost all AI problems. We find search in natural language understanding and generation, in machine vision, in planning and problem solvers, in expert systems, in game playing programs, and in machine learning. It should not be too surprising then that much effort has been devoted to finding efficient search strategies.

Search strategies can be compared by their time and space complexities using big O notation. It is important to determine the complexity of a given strategy before investing too much programming effort, since many search problems are intractable. Search spaces are usually represented as a graph or tree structure, and a search is finding a path from some start node to a goal node.

In a blind search, nodes in the space are explored mechanically until a goal is found, a time limit has been reached, or failure occurs. In a worst case, it may be necessary to explore the whole space before finding a solution. Examples of blind searches are depth-first, breadth-first, and depth-first iterative deepening searches.

When some information is available about the goals, the problem states, or the problem in general, it may be possible to guide the search process and eliminate a number of implausible paths. This is the case in informed searches where cost or another function is used to select the most promising path at each point in the

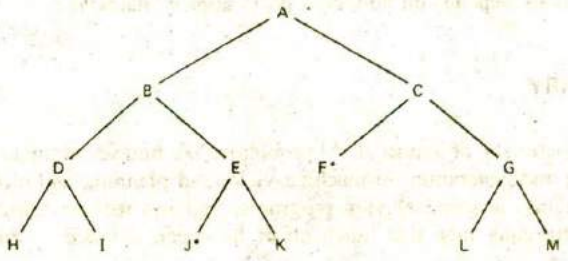
search. Heuristic evaluation functions are used in best-first search strategies to find good solution paths. A solution is not always guaranteed with this type of search, but in most practical cases, good or acceptable solutions are often found.

We saw several examples of informed searches, including general best-first, hill climbing, branch-and-bound, A*, and finally, the optimal And-Or heuristic search known as the OA* algorithm. Desirable properties of heuristic search methods were also defined.

EXERCISES

9.1. Games and puzzles are often used to describe search problems because they are easy to describe. One such puzzle is the farmer-fox-goose-grain puzzle. In this puzzle, a farmer wishes to cross a river taking his fox, goose, and grain with him. He can use a boat which will accommodate only the farmer and one possession. If the fox is left alone with the goose, the goose will be eaten. If the goose is left alone with the grain it will be eaten. Draw a state space search tree for this puzzle using leftbank and rightbank to denote left and right river banks respectively.

9.2. For the search tree given below, use breadth-first searching and list the elements of the queue just before selecting and expanding each next state until a goal node is reached. (Goal states designated with *.)



9.3. Repeat Problem 9.2 using a depth-first search.

9.4. Repeat Problem 9.2 using a depth-first iterative deepening search.

9.5. Describe and compare three primary uninformed search methods described in this chapter.

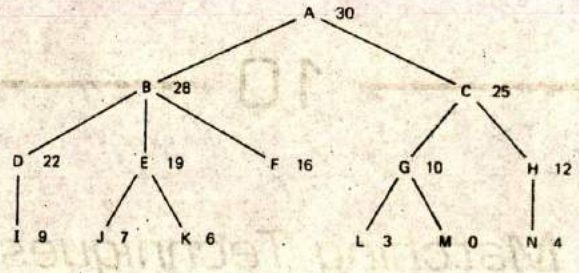
9.6. Show that a worst-case algorithm to solve the traveling salesman problem is of exponential complexity, but an algorithm that chooses a tour through the nearest neighbor of each city is of lower order. Give an example to show the nearest neighbor algorithm is not, in general, optimal, but still often good.

9.7. Fifteen puzzle is like eight puzzle except there are fifteen tiles instead of eight. What is the branching factor of the search tree for fifteen puzzle?

9.8. Describe a problem for which means-end analysis could be successfully applied. Give an example of a few solution steps.

9.9. Give three different heuristics for an $h^*(n)$ to be used in solving the eight puzzle.

9.10. Using the search tree given below, list the elements of the queue just before the next node is expanded. Use best-first search where the numbers correspond to estimated cost-to-goal for each corresponding node.



9.11. Repeat Problem 9.10 when the cost of node B is changed to 18.

9.12. Give the time and space complexities for the search methods of Problems 9.2 and 9.3.

9.13. Discuss some of the potential problems when using hill climbing search. Give examples of the problems cited.

9.14. Discuss and compare hill climbing and best-first search techniques.

9.15. Give an example of an admissible heuristic for the eight puzzle.

9.16. Give two examples of problems in which solutions requiring the minimum search are more appropriate than optimal solutions. Give reasons for your choices.

9.17. Write a LISP program to perform a breadth-first search on a solution space tree constructed using property lists. For example, children nodes e, f, and g of node D of the tree would be constructed with the LISP function

```
(putprop 'D '(E F G) 'children)
```

9.18. Write a LISP program to perform a depth-first search on the tree constructed in Problem 9.17.

10

Matching Techniques

Matching is a basic function that is required in almost all AI programs. It is an essential part of more complex operations such as search and control. In many programs it is known that matching consumes a large fraction of the processing time. For example, it has been estimated that matching operations in many production systems account for as much as 90% of the total computation time. Consequently, the AI practitioner will find it is essential to learn efficient matching techniques. In this chapter we examine such techniques and their application to different AI programs.

10.1 INTRODUCTION

Matching is the process of comparing two or more structures to discover their likenesses or differences. The structures may represent a wide range of objects including physical entities, words or phrases in some language, complete classes of things, general concepts, relations between complex entities, and the like. The representations will be given in one or more of the formalisms like FOPL, networks, or some other scheme, and matching will involve comparing the component parts of such structures.

Matching is used in a variety of programs for different reasons. It may serve to control the sequence of operations, to identify or classify objects, to determine

the best of a number of different alternatives, or to retrieve items from a data base. It is an essential operation in such diverse programs as speech recognition, natural language understanding, vision, learning, automated reasoning, planning, automatic programming, and expert systems, as well as many others.

In its simplest form, matching is just the process of comparing two structures or patterns for equality. The match fails if the patterns differ in any aspect. For example, a match between the two character strings acdebfba and acdebeba fails on an exact match since the strings differ in the sixth character positions.

In more complex cases the matching process may permit transformations in the patterns in order to achieve an equality match. The transformation may be a simple change of some variables to constants, or it may amount to ignoring some components during the match operation. For example, a pattern matching variable such as ?x may be used to permit successful matching between the two patterns (a b (c d) e) and (a b ?x e) by binding ?x to (c d). Such matchings are usually restricted in some way, however, as is the case with the unification of two clauses where only consistent bindings are permitted. Thus, two patterns such as

(a b (c d) e f) and (a b ?x e ?x)

would not match since ?x could not be bound to two different constants.

In some extreme cases, a complete change of representational form may be required in either one or both structures before a match can be attempted. This will be the case, for example, when one visual object is represented as a vector of pixel gray levels and objects to be matched are represented as descriptions in predicate logic or some other high level statements. A direct comparison is impossible unless one form has been transformed into the other.

In subsequent chapters we will see examples of many problems where exact matches are inappropriate, and some form of partial matching is more meaningful. Typically in such cases, one is interested in finding a best match between pairs of structures. This will be the case in object classification problems, for example, when object descriptions are subject to corruption by noise or distortion. In such cases, a measure of the degree of match may also be required.

Other types of partial matching may require finding a match between certain key elements while ignoring all other elements in the pattern. For example, a human language input unit should be flexible enough to recognize any of the following three statements as expressing a choice of preference for the low-calorie food item.

I prefer the low-calorie choice.
I want the low-calorie item.
The low-calorie one please.

Recognition of the intended request can be achieved by matching against key words in a template containing "low-calorie" and ignoring other words except, perhaps, negative modifiers.

Finally, some problems may obviate the need for a form of fuzzy matching where an entity's degree of membership in one or more classes is appropriate. Some classification problems will apply here if the boundaries between the classes are not distinct, and an object may belong to more than one class.

Figure 10.1 illustrates the general match process where an input description is being compared with other descriptions. As stressed earlier, the term *object* is used here in a general sense. It does not necessarily imply physical objects. All objects will be represented in some formalism such as a vector of attribute values, propositional logic or FOPL statements, rules, frame-like structures, or other scheme. Transformations, if required, may involve simple instantiations or unifications among clauses or more complex operations such as transforming a two-dimensional scene to a description in some formal language. Once the descriptions have been transformed into the same schema, the matching process is performed element-by-element using a relational or other test (like equality or ranking). The test results may then be combined in some way to provide an overall measure of similarity. The choice of measure will depend on the match criteria and representation scheme employed.

The output of the matcher is a description of the match. It may be a simple yes or no response or a list of variable bindings, or as complicated as a detailed annotation of the similarities and differences between the matched objects.

To summarize then, matching may be exact, used with or without pattern variables, partial, or fuzzy, and any matching algorithm will be based on such factors as

Choice of representation scheme for the objects being matched,

Criteria for matching (exact, partial, fuzzy, and so on),

Choice of measure required to perform the match in accordance with the chosen criteria, and

Type of match description required for output.

In the remainder of this chapter we examine various types of matching problems and their related algorithms. We begin with a description of representation structures

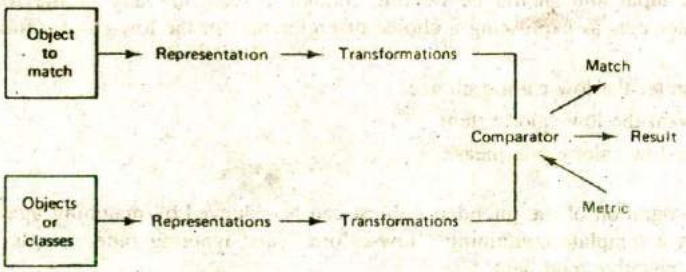


Figure 10.1 Typical matching process.

and measures commonly found in matching problems. We next look at various matching techniques based on exact, partial, and fuzzy approaches. We conclude the chapter with an example of an efficient match algorithm used in some rule-based expert systems.

10.2 STRUCTURES USED IN MATCHING

We are already familiar with many of the representation structures used in matching programs. Typically, they will be some type of list structures that represent clauses in propositional or predicate logic such as

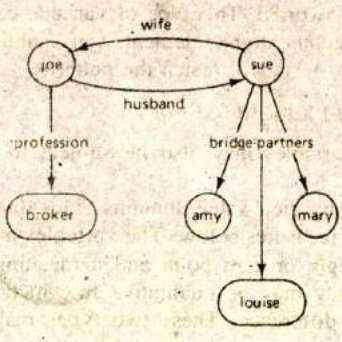
```
(or '(MARRIED ?x ?y) '(DAUGHTER ?z ?y) (MOTHER ?y ?z)),
```

or rules, such as

```
(and ((cloudy-sky) (low-bar-pressure) (high-humidity))
     (conclude (rain-likely)),
```

or fragments of associative networks (Figure 10.2) and frames or frame-like structures (Figure 10.3).

In addition to these, other common structures include strings of characters $a_1a_2 \ldots a_k$, where the a_i belong to a given alphabet A, vectors $\mathbf{X} = (x_1, x_2, \ldots, x_n)$, where the x_i represents attribute values, matrices $\mathbf{M}$ (rows of vectors), general graphs, trees, and sets.




```
(putprop 'joe '(sue) 'wife)
(putprop 'sue '(joe) 'husband)
(putprop 'joe '(broker) 'profession)
(putprop 'sue '(amy louise mary) 'bridge-partners)
```

Figure 10.2 Fragment of associative network and corresponding LISP code

```
          name: data-structures
           ako: university-course
    department: computer-science
       credits: 3-hours
prerequisites:(if-needed check catalog)
```

(a)

```
(data-structures (ako (value university-course))
        (department (value computer-science))
        (credits (value 3-hours))
        (prerequisites (:if-needed check-catalog))))
```

(b)

Figure 10.3 (a) Frame structure and (b) corresponding A-list code.

Variables

All of the structures we shall consider here are constructed from basic atomic elements, numbers, and characters. Character string elements may represent either constants or variables. If variables, they may be classified by either the type of match permitted or by their value domains.

We can classify match variables by the number of items that can replace them (one or more than one). An *open variable* can be replaced by a single item, while a *segment variable* can be replaced by zero or more items. Open variables are labeled with a preceding question mark (?x, ?y, ?class). They may match or assume the value of any single string element or word, but they are sometimes subject to consistency constraints. For example, to be consistent, the variable ?x can be bound only to the same top level element in any single structure. Thus (*a* ?*x d* ?*x e*) may match (*a b d b e*), but not (*a b d a e*). Segment variable types will be preceded with an asterisk (*x, *z, *words). This type of variable can match an arbitrary number or segment of contiguous atomic elements (any sublist including the empty list). For example, (**x d* (*e f*) **y*) will match the patterns

$$(a\ (b\ c)\ d\ (e\ f)\ g\ h),\ (d\ (e\ f)\ (g))$$

or other similar patterns. Segment variables may also be subject to consistency constraints similar to open variables.

Variables may also be classified by their value domains. This distinction will be useful when we consider similarity measures below. The variables may be either quantitative, having a meaningful origin or zero point and a meaningful interval difference between two values, or they may be qualitative in which there is no origin nor meaningful interval value difference. These two types may be further subdivided as follows.

Nominal variables. Qualitative variables whose values or states have no order nor rank. It is only possible to distinguish equality or inequality between two

such objects. Of course each state can be given a numerical code. For example, "marital status" has states of married, single, divorced, or widowed. These states have no numerical significance, and no particular order nor rank. The states could be assigned numerical codes however, such as married = 1, single = 2, divorced = 3, and widowed = 4.

Ordinal variables. Qualitative variables whose states can be arranged in a rank order, but the difference between two distinct values has no significance. Ordinal variables may also be assigned numerical values. For example, the states very tall, tall, medium, short, and very short can be arranged in order from tallest to shortest and be assigned an arbitrary scale of 5 to 1. However, the difference between successive values does not necessarily have any quantitative meaning.

Binary variable. Qualitative discrete variables which may assume only one of two values, such as 0 or 1, good or bad, yes or no, high or low.

Interval (metric) variables. Quantitative variables which take on numeric values and for which equal differences between values have the same significance. For example, real numbers corresponding to temperature or integers corresponding to an amount of money are considered as interval variables.

Graphs and Trees

Two other structures we shall consider in this section are graphs and trees. One type of graph we are already familiar with is the associative network (Chapter 6). Such structures provide a rich variety of representation schemes. More generally, a graph $G = (V, E)$ is an ordered pair of sets V and E. The elements of V are nodes or vertices and the elements of E are a subset of $V \times V$ called edges (or arcs or links). An edge joints two distinct vertices in V.

Directed graphs, or digraphs, have directed edges or arcs with arrows. If an arc is directed from node n_i to n_j, node n_i is said to be a parent or successor of n_j, and n_j is the child or successor of n_i. Undirected graphs have simple edges without arrows connecting the nodes. A path is a sequence of edges connecting two nodes where the endpoint of one edge is the start of its successor. A cycle is a path in which the two end points coincide. A connected graph is a graph for which every pair of vertices is joined by a path. A graph is complete if every element of $V \times V$ is an edge.

A tree is a connected graph in which there are no cycles, and each node has, at most, one parent. A node with no parent is called the root node, and nodes with no children are called leaf nodes. The *depth* of the root node is defined as zero. The depth of any other node is defined to be the depth of its parent plus 1. Pictorial representations of some graphs and a tree are given in Figure 10.4.

Recall that graph representations typically use labeled nodes and arcs where

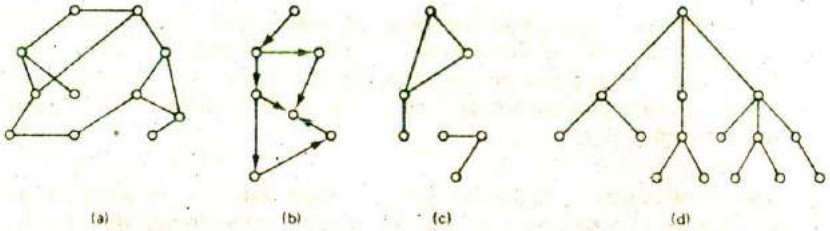


Figure 10.4 Examples of (a) general connected graph, (b) digraph, (c) disconnected graph and (d) tree of depth 3.

the nodes correspond to entities and the arcs to relations. Labels for the nodes and arcs are attribute values.

Sets and Bags

A set is represented as an unordered list of unique elements such as the set (*a d f c*) or (black red blue green). A bag is a set which may contain more than one copy of the same member; for example, the list *a d e a b d* represents a bag with members *a*, *b*, *d*, and *e*. Sets and bags are structures frequently used in matching operations.

10.3 MEASURES FOR MATCHING

Next, we turn to the problem of comparing structures without the use of pattern matching variables. This requires consideration of measures used to determine the likeness or similarity between two or more structures. The similarity between two structures is a measure of the degree of associaton or likeness between the object's attributes and other characteristic parts. If the describing variables are quantitative, a distance metric is often used to measure the proximity.

Distance Metrics

For all elements x, y, z, of the set E, the function d is a metric if and only if

$$\begin{aligned} &\text{a. } d(x,x) = 0 \\ &\text{b. } d(x,y) \geq 0 \\ &\text{c. } d(x,y) = d(y,x) \\ &\text{d. } d(x,y) \leq d(x,z) + d(z,y) \end{aligned} \tag{10.1}$$

The Minkowski metric is a general distance measure satisfying the above assumptions. It is given by

$$d_p = \left[\sum_i^n |x_i - y_i|^p\right]^{1/p}$$

For the case $p = 2$, this metric is the familiar Euclidean distance. When $p = 1$, d_p is the so-called absolute or city block distance.

Probabilistic Measures

In some cases, the representation variables should be treated as random variables. Then one requires a measure of the distance between the variates, their distributions, or possibly between a variable and distribution. One such measure is the Mahalanobis distance which gives a measure of the separation between two distributions. Given the random vectors **X** and **Y** let C be their covariance matrix. Then the Mahalanobis distance is given by

$$D = \mathbf{X}'C^{-1}\mathbf{Y}$$

where the prime (') denotes transpose (row vector) and C^{-1} is the inverse of C. The **X** and **Y** vectors may be adjusted for zero means by first subtracting the vector means u_x and u_y.

Another popular probability measure is the product moment correlation r, given by

$$r = \frac{\text{Cov}(\mathbf{X},\mathbf{Y})}{[\text{Var}(\mathbf{X})*\text{Var}(\mathbf{Y})]^{1/2}}$$

where Cov and Var denote covariance and variance respectively. The correlation r, which ranges between -1 and $+1$, is a measure of similarity frequently used in vision applications.

Other probabilistic measures often used in AI applications are based on the scatter of attribute values. These measures are related to the degree of clustering among the objects. In addition, conditional probabilities are sometimes used. For example, they may be used to measure the liklihood that a given **X** is a member of class C_i, $P(C_i|\mathbf{X})$, the conditional probability of C_i given an observed **X**. These measures can establish the proximity of two or more objects. These and related measures are discussed further in Chapter 12.

Qualitative Measures

A number of distance measures based on qualitative variables (nominal, ordinal, and binary) have also been defined as well as methods which deal with mixtures of variables (Anderberg, 1973). We describe only a few such measures here to illustrate the basic forms they take.

Measures between binary variables are best described using contingency tables like Table 10.1. The table entries there give the number of objects having attribute X or Y with corresponding value of 1 or 0. For example, if the objects are animals,

TABLE 10.1 CONTINGENCY TABLE FOR BINARY VARIABLES

	Variable X	1	0	Totals
Variable Y	1	a	b	a + b
	0	c	d	c + d
	Totals	a + c	b + d	n

X might be horned and Y might be long tailed. In this case, the entry a is the number of animals having both horns and long tails. Note that $n = a + b + c + d$, the total number of objects.

Various measures of association for such binary variables have been defined. For example

$$\frac{a}{a+b+c+d} = \frac{a}{n}, \qquad \frac{a+d}{n}$$

$$\frac{a}{a+b+c}, \qquad \frac{a}{b+c}$$

Contingency tables are also useful for describing other qualitative variables, both ordinal and nominal. Since the methods are similar to those for binary variables, we omit the details here.

Whatever the variable types used in a measure, they should all be properly scaled or normalized to prevent variables having large values from negating the effects of smaller valued variables. This could happen when one variable is scaled in millimeters and another variable in meters.

Similarity Measures

For many problems, distance metrics are not appropriate. Instead, a measure of similarity satisfying conditions different from those of Table 10.1 may be more appropriate. Of course, measures of dissimilarity (or similarity), like distance, should decrease (or increase) as objects become more alike. There is strong evidence, however, to suggest that similarities are not in general symmetric (Tversky, 1977) and hence, any similarity measure between a subject description A and its referrent B, denoted by $s(A,B)$, is not necessarily equal; that is, in general, $s(A,B) \neq s(B,A)$ or "A is like B" may not be the same as "B is like A."

Tests on subjects have shown that in similarity comparisons, the focus of attention is on the subject and, therefore, subject features are given higher weights than the referrent. For example, in tests comparing countries, statements like "North Korea is similar to Red China" and "Red China is similar to North Korea" or "the

USA is like Mexico" and "Mexico is like the USA" were not rated as symmetrical or equal. The likenesses and differences in these cases are directional. Moreover, like many interpretations in AI, similarities may depend strongly on the context in which the comparisons are made. They may also depend on the purpose of the comparison.

An interesting family of similarity measures which takes into account such factors as asymmetry and has some intuitive appeal has recently been proposed (Tversky, 1977). Such measures may be adapted to give more realistic results for similarity measures in AI applications where context and purpose should influence the similarity comparisons.

Let $O = \{o_1, o_2, \ldots\}$ be the universe of objects of interest and let A_i be the set of attributes or features used to represent o_i. A similarity measure s which is a function of three disjoint sets of attributes common to any two objects A_i and A_j is given as

$$s(A_i, A_j) = F(A_i \,\&\, A_j, A_i - A_j, A_j - A_i) \tag{10.2}$$

where $A_i \,\&\, A_j$ is the set of features common to both o_i and o_j, $A_i - A_j$ is the set of features belonging to o_i and not o_j, and $A_j - A_i$ is the set of features belonging to o_j and not o_i. The function F is a real valued nonnegative function. Under fairly general assumptions equation 10.2 can be written as

$$s(A_i, A_j) = af(A_i \,\&\, A_j) - bf(A_i - A_j) - cf(A_j - A_i) \tag{10.3}$$

for some $a, b, c \geq 0$ and where f is an additive interval metric function. The function $f(A)$ may be chosen as any nonnegative function of the set A, like the number of attributes in A or the average distance between points in A. Equation 10.3 may be normalized to give values of similarity ranging between 0 and 1 by writing

$$S(A_i, A2_j) = \frac{f(A_i \,\&\, A_j)}{f(A_i \,\&\, A_j) + af(A_i - A_j) + bf(A_j - A_i)} \tag{10.4}$$

for some $a, b \geq 0$.

When the representations are graph structures, a similarity measure based on the cost of transforming one graph into the other may be used. For example, a procedure to find a measure of similarity between two labled graphs (described in Section 10.5) decomposes the graphs into basic subgraphs and computes the minimum cost to transform either graph into the other one, subpart-by-subpart.

Fuzzy Measures

Finally, we can define a distance between the two fuzzy sets $\tilde{A}$ and $\tilde{B}$ as

$$d(\tilde{A}, \tilde{B}) = \frac{1}{n}\left[\sum_{i=1}^{n} (u_{\tilde{A}}(x_i) - u_{\tilde{B}}(x_i))^2\right]^{1/2} \tag{10.5}$$

where the total number of objects x_i in the universe is n.

Equation 10.5 measures the mean square difference in universe membership scores between $\tilde{A}$ and $\tilde{B}$. A score of 1 implies that the distance is maximal, whereas a score near zero can signify that either $\tilde{A}$ and $\tilde{B}$ are very similar or that most objects have low membership scores in both fuzzy sets. Therefore, care must be exercised when interpreting such measures.

Fuzzy similarity measures can be defined with inverse functions of fuzzy distances such as the measure defined in equation 10.5 above. An even simpler measure is, of course, the characteristic function u itself, which may be regarded as a measure of the distance of an object x_i with respect to some reference fuzzy set. In that case, distance is interpreted as a function of the inverse of degree of membership.

Rather than the distance or similarity between two sets or a set and an object, we may be interested in the distance or similarity between objects (x_i) themselves, where the fuzzy sets represent traits or characteristics of the objects. One such similarity measure for this is

$$s(x_i, x_j) = \frac{1}{K} \sum_{k=1}^{K} (1 - |u_A(x_i) - u_A(x_j)|) \tag{10.6}$$

which gives the mean trait membership difference between two objects x_i and x_j. Of course $s(x_i, x_j) = 0$ corresponds to equal likeness or maximal similarity, and $s(x_i, x_j) = 1$ for $i \neq j$ corresponds to maximum dissimilarity.

10.4 MATCHING LIKE PATTERNS

In this section we consider procedures which amount to performing a complete match between two structures. The match will be accomplished by comparing the two structures and testing for equality among the corresponding parts. Pattern variables will be used for instantiations of some parts subject to restrictions as noted below.

Matching Substrings

Since many of the representation structures are just character strings, a basic function required in many match algorithms is to determine if a substring S_2 consisting of m characters occurs somewhere in a string S_1 of n characters, $m \leq n$. A direct approach to this problem is to compare the two strings character-by-character, starting with the first characters of both S_1 and S_2. If any two characters disagree, the process is repeated, starting with the second character of S_1 and matching again against S_2 character-by-character until a match is found or disagreement occurs again. This process continues until a match occurs or S_1 has no more characters.

Let i and j be position indices for string S_1 and k a position index for S_2. We can perform the substring match with the following algorithm.

```
i:=0
while i≤(n-m+1) do
begin
  i:=i+1; j:=i; k:=1;
  while S1(j)=S2(k) do
  begin
    if k=m writeln('success')
    else do
     begin
       j:=j+1; k:=k+1
     end
   end
  end
 writeln('fail')
end.
```

This algorithm requires $m(n - m)$ comparisons in the worst case. A more efficient algorithm will not repeat the same comparisons over and over again. One such algorithm uses two indices, i and j, where i indexes (counts) the character positions in S_1 and j is set to a "match state" value ranging from 0 to m (like the states in a finite automaton). The state 0 corresponds to no matched characters between the strings, while the state 1 corresponds to the first letter in S_2 matching character i in S_2. State 2 corresponds to the first two consecutive letters in S_2 matching letters i and $i + 1$ in S_1 respectively, and so on, with state m corresponding to a successful match. Whenever consecutive letters fail to match, the state index is reduced accordingly. We leave the actual details as an exercise.

Matching Graphs

Two graphs G_1 and G_2 match if they have the same labeled nodes and same labeled arcs and all node-to-node arcs are the same. More generally, we wish to determine if G_2 with m nodes is a subgraph of G_1 with n nodes, where $n \geq m$. In a worst-case match, this will require $n!/(n - m)!$ node comparisons and $O(m^2)$ arc comparisons. Consequently, we will see that most graph matching applications deal with small manageable graphs only or use some form of heuristics to limit the number of comparisons.

Finding subgraph isomorphisms is also an important matching problem. An isomorphism between the graphs G_1 and G_2 with vertices (nodes) $V1$, $V2$ and edges $E1$, $E2$, that is, $(V1,E1)$ and $(V2,E2)$, respectively, is a one-to-one mapping to f between $V1$ and $V2$, such that for all $v1 \in V1$, $f(v1) = v2$, and for each arc $e1 \in E1$ connecting $v1$ and $v1'$, there is a corresponding arc $e2 \in E2$ connecting $f(v1)$ and $f(v1')$. An example of an application in which graph isomorphisms are used to determine the similarity between two graphs is given in the next section.

Matching Sets and Bags

An exact match of two sets having the same number of elements requires that their intersection also have that number of elements. Partial matches of two sets can also be determined by taking their intersection. If the two sets have the same number of elements and all elements are of equal importance, the degree of match can be the proportion of the total members which match. If the number of elements differ between the sets, the proportion of matched elements to the minimum of the total number of members can be used as a measure of likeness. When the elements are not of equal importance, weighting factors can be used to score the matched elements. For example, a measure such as

$$s(S1,S2) = \left(\sum_{i=1}^{m} w_i N(a_i)\right)/m \qquad (10.7)$$

could be used, where $w_i = 1$ and $N(a_i) = 1$ if a_i is in the intersection; otherwise it is 0.

An efficient way to find the intersection of two sets of symbolic elements (nonnumeric atoms) in LISP is to work through one set marking each element on the elements property list and then saving all elements from the other list that have been marked. The resultant list of saved elements is the required intersection.

Matching two bags is similar to matching two sets except that counts of the number of occurrences of each element must also be made. For this, a count of the number of occurrences can be used as the property mark for elements found in one set. This count can then be used to compare against a count of elements found in the second set.

Matching to Unify Literals

One of the best examples of nontrivial pattern matching is in the unification of two FOPL literals. Recall the procedure for unifying two literals, both of which may contain variables (see Chapter 4). For example, to unify $P(f(a,x),y,y)$ and $P(x,b,z)$ we first rename variables so that the two predicates have no variables in common. This can be done by replacing the x in the second predicate with u to give $P(u,b,z)$. Next, we compare the two symbol-by-symbol from left to right until a disagreement is found. Disagreements can be between two different variables, a nonvariable term and a variable, or two nonvariable terms. If no disagreement is found, the two are identical and we have succeeded.

If a disagreement is found and both are nonvariable terms, unification is impossible; so we have failed. If both are variables, one is replaced throughout by the other. (After any substitution is made, it should be recorded in a substitution worklist for later use.) Finally, if the disagreement is a variable and a nonvariable term, the variable is replaced by the entire term. Of course, in this last step, replacement is

possible only if the term does not contain the variable that is being replaced. This matching process is repeated until the two are unified or until a failure occurs.

For the two predicates P, above, a disagreement is first found between the term $f(a,x)$ and variable u. Since $f(a,x)$ does not contain the variable u, we replace u with $f(a,x)$ everywhere it occurs in the literal. This gives a substitution set of $\{f(a,x)/u\}$ and the partially matched predicates $P(f(a,x),y,y)$ and $P(f(a,x),b,z)$.

Proceeding with the match, we find the next disagreement pair, y and b, a variable and term, respectively. Again, we replace the variable y with the term b and update the substitution list to get $\{f(a,x)/u, b/y\}$. The final disagreement pair is two variables. Replacing the variable in the second literal with the first we get the substitution set $\{f(a,x)/u,b/y,y/z\}$ or, equivalently, $\{f(a,x)/u,b/y,b/z\}$. Note that this procedure can always give the most general unifier.

We conclude this section with an example of a LISP program which uses both the open and the segment pattern matching variables to find a match between a pattern and a clause.

```
(defun match (pattern clause)
  (cond   ((equal pattern clause) t)                  ;return t if
          ((or (null pattern) (null clause)) nil)     ;equal, nil
                                                      ;if not.
          ((or (equal (car pattern) (car clause))     ;not. ?x
                                                      ;binds
               (equal (car pattern) '?x))             ;to single
           (match (cdr pattern) (cdr clause)))        ;term,*y
                                                      ;binds
          ((equal (car pattern) '*y)                  ;to several
           (or (match (cdr pattern) (cdr clause))     ;contiguous
               (match pattern (cdr clause))))))       ;terms.
```

Notice that when a segment variable is encountered (the *y), match is recursively executed on the cdrs of both pattern and clause or on the cdr of clause and pattern as *y matches one or more than one item respectively.

10.5 PARTIAL MATCHING

For many AI applications complete matching between two or more structures is inappropriate. For example, input representations of speech waveforms or visual scenes may have been corrupted by noise or other unwanted distortions. In such cases, we do not want to reject the input out of hand. Our systems should be more tolerant of such commonly occurring problems. Instead, we want our systems to be able to find an acceptable or best match between the input and some reference description.

Compensating for Distortions

Finding an object in a photograph given only a general description of the object is a common problem in vision applications. For example, the task may be to locate a human face or human body in photographs without the necessity of storing hundreds of specific face templates. A better approach in this case would be to store a single reference description of the object. Matching between photograph regions and corresponding descriptions then could be approached using either a measure of correlation or, alternatively, by altering the image to obtain a closer fit. If nothing is known about the noise and distortion characteristics, correlation methods can be ineffective or even misleading. In such cases, methods based on mechanical distortion may be more appropriate.

Imagine that our reference image is on a transparent rubber sheet. This sheet is moved over the input image and at each location is stretched to get the best match alignment between the two images. The match between the two can then be evaluated by how well they correspond and how much push-and-pull distortion is needed to obtain the best correspondence.

In practice, a discrete version of the stretchable model is needed for computer implementation. One way this can be accomplished is to use a number of rigid pieces (like templates) connected with springs. The pieces can correspond to low level areas such as pixels or even larger area segments (Figure 10.5).

To model any restrictions such as the relative positions of body parts (eyes must be above nose, legs below torso, and so on), nonlinear cost functions of piece displacements can be used. The costs can correspond to different spring tensions which reflect the constraints. For example, the cost of displacing some pieces might be zero for no displacement, one unit for single increment displacements in any one of the permissible directions (left, right, up, down), two units for two position

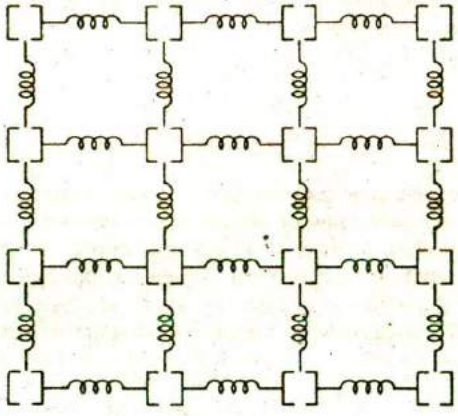


Figure 10.5 Discrete version of stretchable overlay image.

displacements and infinite cost for displacements of more than two increments. Other pieces would be assigned higher costs for unit and larger position displacements when stronger constraints were applicable.

The matching problem here is to find a least cost location and distortion pattern for the reference sheet with regard to the sensed picture. Attempting to compare each component of some reference to each primitive part of a sensed picture is a combinatorially explosive problem. However, in using the template-spring reference image and heuristic methods (based on dynamic programming techniques) to compare against different segments of the sensed picture, the search and match process can be made tractible.

Any matching metric used in the least cost comparison would need to take into account the sum of the distortion costs C_d, the sum of the costs for reference and sensed component dissimilarities C_c, and the sum of penalty costs for missing components C_m. Thus, the total cost is given by

$$C_t = C_d + C_c + C_m \tag{10.8}$$

Finding Match Differences

Distortions occurring in representations are not the only reasons for partial matches. For example, in problem solving or analogical inference, differences are expected. In such cases the two structures are matched to isolate the differences in order that they may be reduced or transformed. Once again, partial matching techniques are appropriate. The problem is best illustrated with another example.

In a vision application (Eshera and Fu, 1984), an industrial part may be described using a graph structure where the set of nodes correspond to rectangular or cylindrical block subparts. The arcs in the graph correspond to positional relations between the subparts. Labels for rectangular block nodes contain length, width, and height, while labels for cylindrical block nodes give radius and height. The arc labels give location and distances between block nodes, where location can be above, to the right of, behind, inside, and so on.

Figure 10.6 illustrates a segment of such a graph. In the figure the following abbreviations are used:

R = rectangular block	l_i = length of subpart
C = cylindrical block	w_i = width of subpart
J = joint	h_i = height of subpart
T = to-the-right-of	r_i = radius of subpart
V = above	d_i = distance between subparts

Interpreting the graph, we see it is a unit consisting of subparts, made up of rectangular and cylindrical blocks with dimensions specified by attribute values. The cylindrical block n_1 is to the right of n_2 by d_1 units and the two are connected by a joint. The blocks n_1 and n_2 are above the rectangular block n_3 by d_2 and d_3 units respectively, and so on.

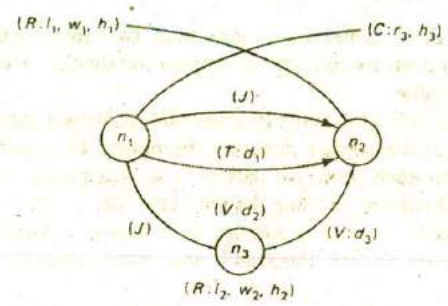


Figure 10.6 Segment of an attributed relational graph.

Graphs such as this are called attributed relational graphs (ATRs). Such a graph G is defined formally as a sextuple

$$G = (N,B,A,G_n,G_b)$$

where $N = \{n_1, n_2, \ldots, n_k\}$ is a set of nodes, $A = \{an_1, an_2, \ldots, an_k\}$ is an alphabet of node attributes, $B = \{b_1, b_2, \ldots, b_m\}$ is a set of directed branches ($b = (n_i,n_j)$), and G_n and G_b are functions for generating node and branch attributes respectively.

When the representations are graph structures like ARGs, a similarity measure may be computed as the total cost of transforming one graph into the other. For example, the similarity of two ARGs may be determined with the following steps: (1) Decompose the ARGs into basic subgraphs, each having a depth of one, (2) compute the minimum cost to transform either basic ARG into the other one subgraph-by-subgraph, and (3) compute the total transformation cost from the sum of the subgraph costs.

An ARG may be transformed by the three basic operations of node or branch deletions, insertions, or substitutions, where each operation is given a cost based on computation time or other factors.

Finding the minimal cost to transform one ARG into another is known to be an NP complete problem. However, heuristic solutions using dynamic programming methods have been developed which have time complexity of $O(m^2n^2(m + n))$ where m and n are the number of nodes in the two ARGs. For details regarding such computation procedures the reader is referred to Eshera and Fu (1984).

10.6 FUZZY MATCHING ALGORITHMS

Fuzzy matching is accomplished by computing a fuzzy distance or similarity measure between two objects such as those given in Section 10.3. A similarity score of 1 corresponds to an identical match while a score near 0 corresponds to maximal dissimilarity. For example, suppose two objects say o_1 and o_2, are each described by the same set of k attributes A_i, $i = 1, \ldots, k$. Each attribute may be regarded

as a fuzzy set, and a metric similar to equation 10.6 may then be used to match compare the two objects based on their attribute memberships.

If the attributes represent linguistic variables such as height, weight, facial-appearance, color-of-eyes, and type-of-hair, each variable may be assigned a limited number of values. For example, a reasonable assignment for height would be the integers 10 to 96 corresponding to height in inches. Eye colors could be assigned brown, black, blue, hazel, and so on. An object description of tall, slim, pretty, blue-eyed, blonde will have characteristic function values for the five attributes of $u_{A_i}(o_1)$ and $u_{A_i}(o_2)$ for objects o_1 and o_2 respectively. A measure of fuzzy similarity between the two objects can then be defined as

$$s(o_1, o_2) = 1 / (1 - d),$$

where

$$d = \frac{1}{k}\left[\sum_{i=1}^{k}(u_{A_i}(o_1) - u_{A_i}(o_2))^2\right]^{1/2} \tag{10.9}$$

For an accurate match, the quantity d in equation 10.9 should be computed for several different values of each linguistic variable (very short, short, medium, tall, and very tall) and the average taken. This will be done at the expense of much computation, however.

Note that it is always possible to define the attribute domains as discrete finite approximations to any domain as we have done for height, and the characteristic values for tall might be $u_{\text{tall}}(10) = \ldots u_{\text{tall}}(50) = 0, \ldots, u_{\text{tall}}(66) = 0.5, \ldots$.

The number (k) of attributes and domain values chosen for each attribute will depend on the specific application. Furthermore, it should be noted that the characteristic values will be subjective ones. Even so, selecting only a few relevant attributes and assigning a modest number of values for each domain can give a good approximation to fuzzy likenesses.

10.7 THE RETE MATCHING ALGORITHM

Production (or rule-based) systems are described in Chapter 15. They are popular architectures for expert systems. A typical system will contain a Knowledge Base which contains structures representing the domain expert's knowledge in the form of rules or productions, a working memory which holds parameters for the current problem, and an inference engine with rule interpreter which determines which rules are applicable for the current problem (Figure 10.7).

The basic inference cycle of a production system is match, select, and execute as indicated in Figure 10.7. These operations are performed as follows:

Match. During the match portion of the cycle, the conditions in the left hand side (LHS) of the rules in the knowledge base are matched against the contents

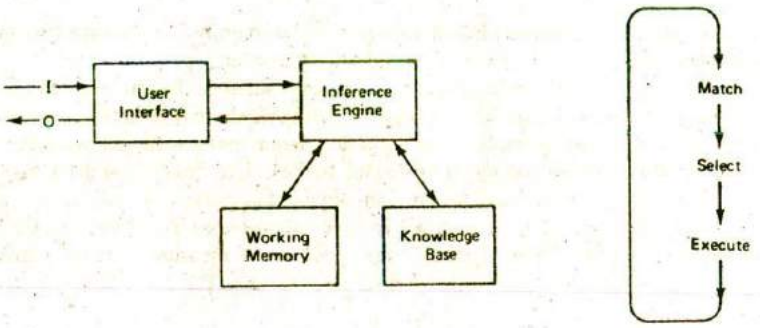


Figure 10.7 Production system components and basic cycle.

of working memory to determine which rules have their LHS conditions satisfied with consistent bindings to working memory terms. Rules which are found to be applicable (that match) are put in a conflict set.

Select. From the conflict set, one of the rules is selected to execute. The selection strategy may depend on recency of useage, specificity of the rule, or other criteria.

Execute. The rule selected from the conflict set is executed by carrying out the action or conclusion part of the rule, the right hand side (RHS) of the rule. This may involve an I/O operation, adding, removing or changing clauses in Working Memory or simply causing a halt.

The above cycle is repeated until no rules are put in the conflict set or until a stopping condition is reached.

A typical knowledge base will contain hundreds or even thousands of rules and each rule will contain several (perhaps as many as ten or more) conditions. Working memories typically contain hundreds of clauses as well. Consequently, exhaustive matching of all rules and their LHS conditions against working-memory clauses may require tens of thousands of comparisons. This accounts for the claim made in the introductory paragraph that as much as 90% of the computing time for such systems can be related to matching operations.

To eliminate the need to perform thousands of matches per cycle, an efficient match algorithm called RETE has been developed (Forgy, 1982). It was initially developed as part of the OPS family of programming languages (Brownston, et al., 1985). This algorithm uses several novel features, including methods to avoid repetitive matching on successive cycles. The main time-saving features of RETE are as follows.

1. In most expert systems, the contents of working memory change very little from cycle to cycle. There is a persistence in the data known as temporal redundancy.

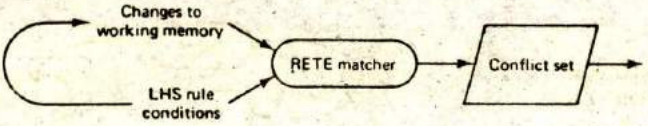


Figure 10.8 Changes to working memory are mapped to the conflict set.

This makes exhaustive matching on every cycle unnecessary. Instead, by saving match information, it is only necessary to compare working memory changes on each cycle. In RETE, additions to, removals from, and changes to working memory are translated directly into changes to the conflict set (Figure 10.8). Then, when a rule from the conflict set has been selected to fire, it is removed from the set and the remaining entries are saved for the next cycle. Consequently, repetitive matching of all rules against Working Memory is avoided. Furthermore, by indexing rules with the condition terms appearing in their LHS (described below), only those rules which could match Working Memory changes need to be examined. This greatly reduces the number of comparisons required on each cycle.

2. Many rules in a knowledge base will have the same conditions occurring in their LHS. This is just another way in which unnecessary matching can arise. Repeated testing of the same conditions in those rules could be avoided by grouping rules which share the same conditions and linking them to their common terms. It would then be possible to perform a single set of tests for all the applicable rules. A description of this linking process is given below.

When the rules are first loaded into the knowledge base, they are examined and processed by a rule compiler. The compiler checks the LHS conditions and forms an association between rule names and their LHS condition terms. In addition, the compiler builds a network structure which connects all rules having common conditions in their LHS. The network is then used during run time to locate and test rule conditions which might be satisfied with consistent bindings to new working-memory clauses. Figure 10.9 illustrates how rules sharing common LHS terms are grouped together and indexed with these common condition terms.

One way to form the associations and indices using LISP is with property lists. For example,

```
(putprop 'R6 'father 'cond-1)
(putprop 'R6 'father 'cond-2)
(putprop 'R12 'father 'cond-1)
```

sets up a link between rules and their LHS conditions, whereas statements like

```
(putprop 'father (cons R6 (get 'father 'cond-1) 'cond-1))
```

link specific LHS terms to all rules which contain the term in the same LHS positions. When a change is made to working memory, such as the addition of the clause

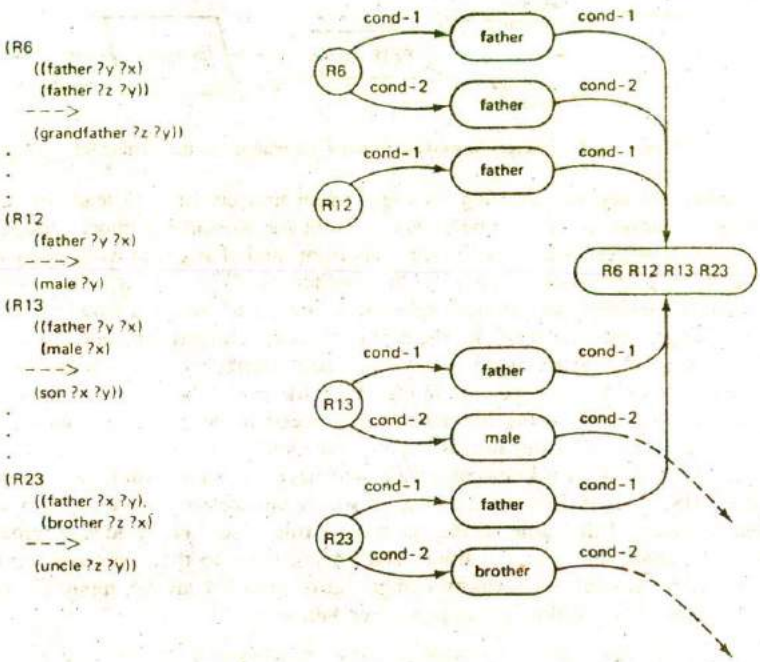


Figure 10.9 Typical rules and a portion of a compiled network.

(father bill joe), all rules which contain father as an LHS condition are easily identified and retrieved.

In RETE, the retrieval and subsequent testing of rule conditions is initiated with the creation of a token which is passed to the network constructed by the rule compiler. The network provides paths for all applicable tests which can lead to consistent bindings and hence to complete LHS satisfaction of rules. The matcher traverses the network finding all rules which newly match or no longer match Working Memory elements. The output from the matcher are data structures which consist of pairs of elements: a rule name and list of working-memory elements that match its LHS, like (R6 ((father bob sam) (father mike bob)).

The reader will notice that the indexing methods described above are similar to those presented in the following chapter. Other time-saving tricks are also employed in RETE; however, the ones noted above are the most important. They provide a substantial saving over exhaustive matching of hundreds or even tens of thousands of conditions.

10.8 SUMMARY

In this chapter we have examined representation structures used in match comparison operations, and considered various measures of similarity and distance between two or more such structures.

We began by reviewing some representation structures such as FOPL expressions, rules, frames, fuzzy sets, and networks. We added to this group general graphs, trees, sets, and bags. We defined pattern variables as open or segment based on the number of constant terms they could bind with. We also defined variables as a function of their domains, nominal, ordinal, binary, and interval.

Next, we considered various measures that could be used in assessing the likeness or proximity of two or more objects. These included Euclidean distance, probabilistic measures, qualitative measures, various similarity measures, and fuzzy measures.

We then examined matching algorithms for exact matches where the structures were required to be identical, be transformable, or be capable of binding to pattern variables with certain constraints.

We also considered partial matching problems and saw two examples in which partial matches were more realistic than exact matches. Fuzzy matching procedures for objects were also described and appropriate measures presented.

Finally, we concluded the chapter with a description of an important matching algorithm used in the OPS programming languages. This is the RETE algorithm which takes advantage of the fact that the contents of working memory change little from cycle to cycle, and many rules share the same conditions in their LHS. By properly indexing predicates and rules and saving match information, RETE is able to eliminate exhaustive matching on every cycle and update the conflict set only as needed.

EXERCISES

10.1. Indicate whether or not consistent substitutions can be made which result in matches for the following pairs of clauses. If substitutions can be made, given examples of valid ones.

a. P(a,f(x,b),g(f(a,y)),z), P(a,f(a,y),g(f(x,y)),c)
b. P(a,x) V Q(b,y,f(y)) V R(x,y),
P(x,a) V Q(f(y),y,b) V R(y,x)
c. R(a,b,c) V Q(x,y,z) V P(f(a,x,b),
P(z) V Q(x,y,b) V R(x,y,z)

10.2. State what variable bindings, if any, will make the following lists match.

a. (a b c (d a) f), (?x b c (d ?y) ?z)
b. (*x a b (c d) *x), ((e f) a b *y e f)
c. (?x *y a b c (d e)), (a (j k (f)) a b c *z)

10.3. Write a LISP function called "match" that takes two arguments and returns T if the two are identical, returns the two arguments if one is a variable and the other a term and returns nil, otherwise.

10.4. Identify the following variables as nominal, ordinal, binary or interval:

temperature	sex
wavelength	university class
population	intelligence
quality of restaurant	

10.5. What is the difference between a bag and a set? Give examples of both. How could a program determine whether a data structure was either a bag or a set?

10.6. Compute the Mahalanobis distance between two normal distributions having zero means, variances of 4 and 9, and a covariance of 5.

10.7. Give three different examples of functions f that can be used in the similarity equations 10.3 and 10.4.

10.8. Choose two simple objects $O1$ and $O2$ that are somewhat similar in their features $A1$ and $A2$, respectively, and compute the similarity of the two using a form of equation 10.4.

10.9. Define two fuzzy sets "tall" and "short" and compute the distance between them using equation 10.5.

10.10. For the two sets defined in Problem 10.9, compute the similarity of the two using equation 10.6.

10.11. Write a LISP function to find the intersection of two sets using the marking method described in the subsection entitled Matching Sets and Bags.

10.12. Write a LISP function that determines if two sets match exactly.

10.13. Write pseudocode to unify two FOPL literals.

10.14. Write a LISP program based on the pseudocode developed in Problem 10.13.

10.15. Write pseudocode to find the similarity between two attributed relational graphs (AGRs).

10.16. Suppose an expert system working memory has m clauses each with an average of four if . . then conditions per clause and a knowledge base with 200 rules. Each rule has an avereage of five conditions. What is the time complexity of a matching algorithm which performs exhaustive matching?

10.17. Estimate the average time savings if the RETE algorithm was used in the previous problem.

10.18. Write a PROLOG program that determines if two sets match exactly.

10.19. Write a PROLOG program that determines if two sets match except possibly for the first elements of each set.

11

Knowledge Organization and Management

We have seen how important the choice of a suitable representation can be in the solution of knowledge-based problems. When a good representation is chosen for a class of problems, the solution process can be greatly simplified. A poor representation can lead to excessive effort or even failure. Another factor which can have a significant impact on the ease with which problems are solved is the accessibility of the knowledge. By accessibility, we mean the ease and the reliability with which a specific set of knowledge can be selectively found and retrieved for use over extended periods of time.

The problem of access is closely related to and dependent on the way in which the knowledge is organized and maintained in memory. Through appropriate structuring of the knowledge, the retrieval process can be greatly expedited. But a memory organization for an intelligent system must not be a static one. In order to be effective, it must be dynamic, since knowledge will continually change with modifications to the environment. New knowledge must be integrated with the old, and outmoded knowledge must be modified or forgotten. This requires continual reorganization of the knowledge.

Knowledge-based systems tend to require large amounts of knowledge. And, as knowledge bases increase in size and complexity, the access problem becomes more difficult. The time to search, test, select, and retrieve a minimal amount of requisite knowledge from a large body of knowledge can be very time consuming

if the knowledge is poorly organized. Such problems can easily become intractible or at best intolerable.

In this chapter, we investigate various approaches to the effective organization of knowledge within memory. We recognize that while the representation of knowledge is still an important factor, we are more concerned here with the broader problem, that of organization and maintenance for efficient storage and recall as well as for its manipulation.

11.1 INTRODUCTION

The advantages of using structured knowledge representation schemes (frames, associative networks, or object-oriented structures) over unstructured ones (rules or FOPL clauses) should be understood and appreciated at this point. Structured schemes group or link small related chunks of knowledge together as a unit. This simplifies the processing operations, since knowledge required for a given task is usually contained within a limited semantic region which can be accessed as a unit or traced through a few linkages.

But, as suggested earlier, representation is not the only factor which affects efficient manipulation. A program must first locate and retrieve the appropriate knowledge in an efficient manner whenever it is needed. One of the most direct methods for finding the appropriate knowledge is exhaustive search or the enumeration of all items in memory. This is also one of the least efficient access methods. More efficient retrieval is accomplished through some form of indexing or grouping. We consider some of these processes in the next section where we review traditional access and retrieval methods used in memory organizations. This is followed by a description of less commonly used forms of indexing.

A "smart" expert system can be expected to have thousands or even tens of thousands of rules (or their equivalent) in its KB. A good example is XCON (or R1), an expert system which was developed for the Digital Equipment Corporation to configure their customers' computer systems. XCON has a rapidly growing KB which, at the present time, consists of more than 12,000 production rules. Large numbers of rules are needed in systems like this which deal with complex reasoning tasks. System configuration becomes very complex when the number of components and corresponding parameters is large (several hundred). If each rule contained about four or five conditions in its antecedent or If part and an exhaustive search was used, as many as 40,000–50,000 tests could be required on each recognition cycle. Clearly, the time required to perform this number of tests is intolerable. Instead, some form of memory management is needed. We saw one way this problem was solved using a form of indexing with the RETE algorithm described in the preceding chapter. More direct memory organization approaches to this problem are considered in this chapter.

We humans live in a dynamic, continually changing environment. To cope

with this change, our memories exhibit some rather remarkable properties. We are able to adapt to varied changes in the environment and still improve our performance. This is because our memory system is continuously adapting through a reorganization process. New knowledge is continually being added to our memories, existing knowledge is continually being revised, and less important knowledge is gradually being forgotten. Our memories are continually being reorganized to expand our recall and reasoning abilities. This process leads to improved memory performance throughout most of our lives.

When developing computer memories for intelligent systems, we may gain some useful insight by learning what we can from human memory systems. We would expect computer memory systems to possess some of the same features. For example, human memories tend to be limitless in capacity, and they provide a uniform grade of recall service, independent of the amount of information stored. For later use, we have summarized these and other desirable characteristics that we feel an effective computer memory organization system should possess.

1. It should be possible to add and integrate new knowledge in memory as needed without concern for limitations in size.
2. Any organizational scheme chosen should facilitate the remembering process. Thus, it should be possible to locate any stored item of knowledge efficiently from its content alone.
3. The addition of more knowledge to memory should have no adverse effects on the accessibility of items already stored there. Thus, the search time should not increase appreciably with the amount of information stored.
4. The organization scheme should facilitate the recognition of similar items of knowledge. This is essential for reasoning and learning functions. It suggests that existing knowledge be used to determine the location and manner in which new knowledge is integrated into memory.
5. The organization should facilitate the process of consolidating recurrent incidents or episodes and "forgetting" knowledge when it is no longer valid or no longer needed.

These characteristics suggest that memory be organized around *conceptual clusters* of knowledge. Related clusters should be grouped and stored in close proximity to each other and be linked to similar concepts through associative relations. Access to any given cluster should be possible through either direct or indirect links such as concept pointers indexed by meaning. Index keys with synonomous meanings should provide links to the same knowledge clusters. These notions are illustrated graphically in Figure 11.1 where the clusters represent arbitrary groups of closely related knowledge such as objects and their properties or basic conceptual categories. The links connecting the clusters are two-way pointers which provide relational associations between the clusters they connect.

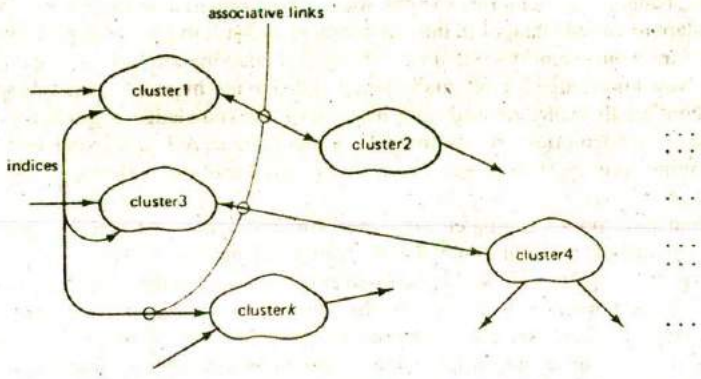


Figure 11.1 Indexed clusters of linked knowledge.

The Frame Problem

One tricky aspect of systems that must function in dynamic environments is due to the so-called frame problem. This is the problem of knowing what changes have and have not taken place following some action. Some changes will be the direct result of the action. Other changes will be the result of secondary or side effects rather than the result of the action. For example, if a robot is cleaning the floors in a house, the location of the floor sweeper changes with the robot even though this

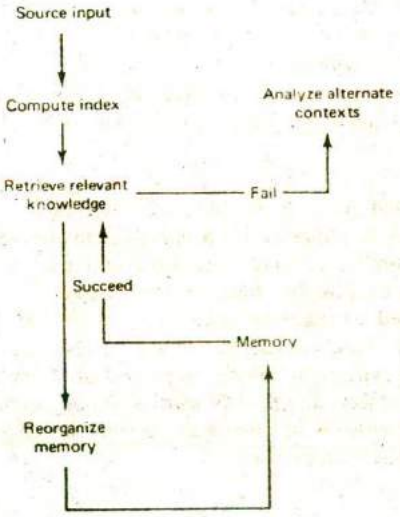


Figure 11.2 Memory organization functions

is not explicitly stated. Other objects not attached to the robot remain in their original places. The actual changes must somehow be reflected in memory, a feat that requires some ability to infer. Effective memory organization and management methods must take into account effects caused by the frame problem.

In the remainder of this chapter we consider three basic problems related to knowledge organization: (1) classifying and computing indices for input information presented to a system, (2) access and retrieval of knowledge from memory through the use of the computed indices, and (3) the reorganization of memory structures when necessary to accommodate additions, revisions, and forgetting. These functions are depicted in Figure 11.2.

11.2 INDEXING AND RETRIEVAL TECHNIQUES

When a knowledge base is too large to be held in main memory, it must be stored as a file in secondary storage (disk, drum or tape). Storage and retrieval of information in secondary memory is then performed through the transfer of equal-size physical blocks consisting of between 2^8 (256) and 2^{12} (4096) bytes. When an item of information is retrieved or stored, at least one complete block must be transferred between main and secondary memory. The time required to transfer a block typically ranges between 10 ms. and 100 ms., about the same amount of time required to sequentially search the whole block for an item. Clearly, then, grouping related knowledge together as a unit can help to reduce the number of block transfers, and hence the total access time.

An example of effective grouping alluded to above, can be found in some expert system KB organizations. Grouping together rules which share some of the same conditions (propositions) and conclusions can reduce block transfer times since such rules are likely to be needed during the same problem-solving session. Consequently, collecting rules together by similar conditions or content can help to reduce the number of block transfers required. As noted before, the RETE algorithm, described in the previous chapter, is an example of this type of organization.

Indexed Organization

While organization by content can help to reduce block transfers, an indexed organization scheme can greatly reduce the time to determine the storage location of an item. Indexing is accomplished by organizing the information in some way for easy access. One way to index is by segregating knowledge into two or more groups and storing the locations of the knowledge for each group in a smaller index file. To build an indexed file, knowledge stored as units (such as records) is first arranged sequentially (sorted) by some key value. The key can be any chosen field or fields that uniquely identify the record. A second file containing indices for the record locations is created while the sequential knowledge file is being loaded. Each physical block in this main file results in one entry in the index file. The index file entries

are pairs of record key values and block addresses. The key value is the key of the first record stored in the corresponding block. To retrieve an item of knowledge from the main file, the index file is searched to find the desired record key and obtain the corresponding block address. The block is then accessed using this address. Items within the block are then searched sequentially for the desired record.

An indexed file contains a list of the entry pairs (k,b) where the values k are the keys of the first record in each block whose starting address is b. Figure 11.3 illustrates the process used to locate a record using the key value of 378. The largest key value less than 378 (375) gives the block address (800) where the item will be found. Once the 800 block has been retrieved, it can be searched linearly to locate the record with key value 378. This key could be any alphanumeric string that uniquely identifies a block, since such strings usually have a collation order defined by their code set.

If the index file is large, a binary search can be used to speed up the index file search. A binary search will significantly reduce the search time over linear search when the number of items is not too small. When a file contains n records, the average time for a linear search is proportional to $n/2$ compared to a binary search time on the order of $\ln_2(n)$.

Further reductions in search time can be realized using secondary or higher order (hierarchically) arranged index files. In this case the secondary index file would contain key and block-address pairs for the primary index file. Similar indexing would apply for higher order hierarchies where a separate file is used for each level. Both binary search and hierarchical index file organization may be needed when the KB is a very large file.

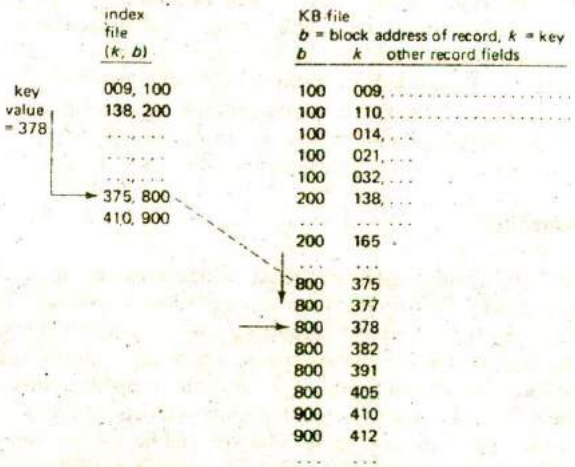


Figure 11.3 Indexed file organization.

When the total number of records in a KB file is n with r records stored per block giving a total of b blocks ($n = r * b$), the average search time for a nonindexed, sequential search is $b / 2$ block access times plus $n / 2$ record tests. This compares with an index search time of $b / 2$ index tests, one block access, and $r / 2$ record tests. A binary index search on the other hand would require only $\ln_2(b)$ index tests, one block access, and $r / 2$ record tests. Therefore, we see that for large n and moderately large r (30 to 80), the time savings possible using binary indexed access can be substantial.

Indexing in LISP can be implemented with property lists, A-lists, and or hash-tables. For example, a KB can be partitioned into segments by storing each segment as a list under the property value for that segment. Each list indexed in this way can be found with the get property function and then searched sequentially or sorted and searched with binary search methods. A hash-table is a special data structure in LISP which provides a means of rapid access through key hashing. We review the hashing process next.

Hashed Files

Indexed organizations that permit efficient access are based on the use of a hash function. A hash function, h, transforms key values k into integer storage location indices through a simple computation. When a maximum number of items or categories C are to be stored, the hashed values $h(k)$ will range from 0 to $C - 1$. Therefore, given any key value k, $h(k)$ should map into one of $0 \ldots C - 1$.

An effective, but simple hash function can be computed by choosing the largest prime number p less than or equal to C, converting the key value k into an integer k' if necessary, and then using the value k' mod p as the index value h. For example, if C is 1000, the largest prime less than C is $p = 997$. Thus, if the record key value is 123456789 (a social security number), the hashed value is $h = (k \bmod 997) = 273$.

When using hashed access, the value of C should be chosen large enough to accommodate the maximum number of categories needed. The use of the prime number p in the algorithm helps to insure that the resultant indices are somewhat uniformly distributed or hashed throughout the range $0 \ldots C - 1$.

This type of organization is well suited for groups of items corresponding to C different categories. When two or more items belong to the same category, they will have the same hashed values. These values are called *synonyms*. One way to accommodate collisions (simultaneous attempts to access synonyms) is with data structures known as buckets. A bucket is a linked list of one or more items, where each item is a record, block, list or other data structure. The first item in each bucket has an address corresponding to the hashed address. Figure 11.4 illustrates a form of hashed memory organization which uses buckets to hold all items with the same hashed key value. The address of each bucket in this case is the indexed location in an array.

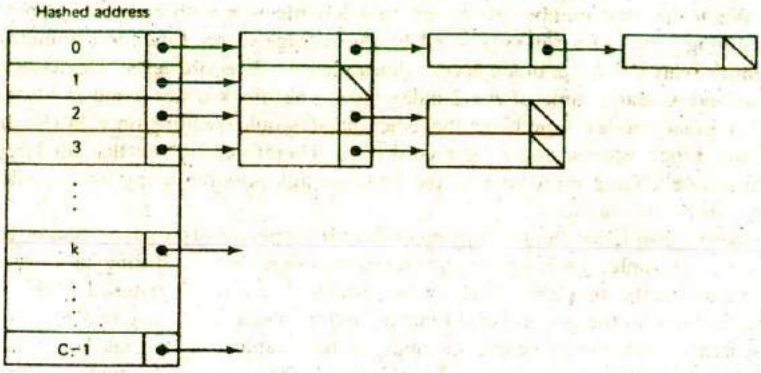


Figure 11.4 Hashed memory file organization.

Conceptual Indexing

The indexing schemes described above are based on lexical ordering, where the collation order of a key value determines the relative location of the record. Keys for these items are typically chosen as a coded field (employee number, name, part number, and so on) which uniquely identifies the item. A better approach to indexed retrieval is one which makes use of the content or meaning associated with the stored entities rather than some nonmeaningful key value. This suggests the use of indices which name and define or otherwise describe the entity being retrieved. Thus, if the entity is an object, its name and characteristic attributes would make meaningful indices. If the entity is an abstract object such as a concept, the name and other defining traits would be meaningful as indices.

How are structures indexed by meaning, and how are they organized in memory for retrieval? One straightforward and popular approach uses associative networks (see Chapter 7) similar to the structures illustrated in Figure 11.1. Nodes within the network correspond to different knowledge entities, whereas the links are indices or pointers to the entities. Links connecting two entities name the association or relationship between them. The relationship between entities may be defined as a hierarchical one or just through associative links.

As an example of an indexed network, the concept of computer science (CS) should be accessible directly through the CS name or indirectly through associative links like a university major, a career field, or a type of classroom course. These notions are illustrated in Figure 11.5.

Object attributes can also serve as indices to locate items or categories based on the attribute values. In this case, the best attribute keys are those which provide the greatest discrimination among objects within the same category. For example, suppose we wish to organize knowledge by object types. In this case, the choice of attributes should depend on the use intended for the knowledge. Since objects

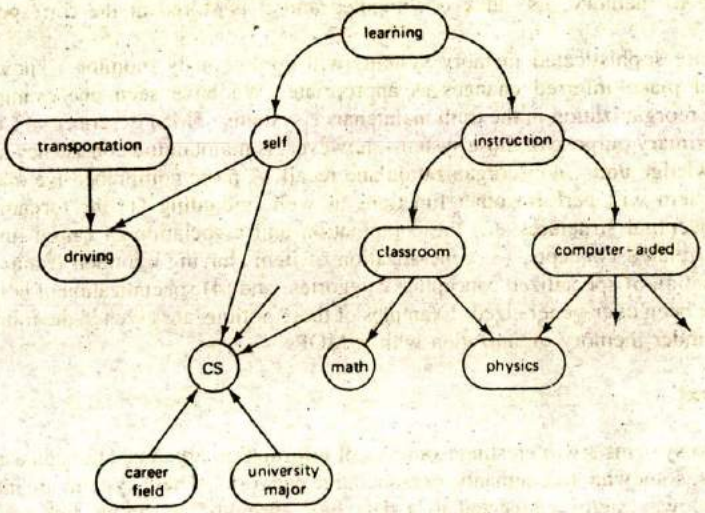


Figure 11.5 Associative network indexing and organization.

may be classified with an unlimited number of attributes (color, size, shape, markings, and so on), those attributes which are most discriminable with respect to the concept meaning should be chosen. Alternatively, object features with the most predictive power make the best indices. A good index for bird types is one based on individual differences like feet, size, beak shape, sounds emitted, special markings, and so forth. Attribute values possessed by all objects are useful for forming categories but poor for identifying an object within the category.

Truly intelligent methods of indexing will be content associative and usually require some inferring. Like humans, a system may fail to locate an item when it has been modified in memory. In such cases, cues related to the item may be needed. For example, you may fail to remember whether or not you have ever discussed American politics with a foreigner until you have considered under what circumstances you may have talked with foreigners (at a university, while traveling or living abroad, or just a chance meeting). An example of this type of indexing strategy is discussed in Section 11.4.

11.3 INTEGRATING KNOWLEDGE IN MEMORY

Integrating new knowledge in traditional data bases is accomplished by simply adding an item to its key location, deleting an item from a key directed location, or modifying fields of an existing item with specific input information. When an item in inventory is replaced with a new one, its description is changed accordingly. When an item

is added to memory, its index is computed and it is stored at the corresponding address.

More sophisticated memory systems will continuously monitor a knowledge base and make inferred changes as appropriate. We have seen one example of memory reorganization in the truth maintenance systems (TMS) described in Chapter 5. The primary purpose of those systems, however, is maintaining consistency among the knowledge units, not reorganization and recall. A more comprehensive management system will perform other functions as well, including (1) the formation of new conceptual structures, (2) the computation and association of causal linkages between related concepts, (3) generalization of items having common features and the formation of specialized conceptual categories, and (4) specialization of concepts that have been over-generalized. Examples of these notions are given in the following section under memory organization with E-MOPs.

Hypertext

Hypertext systems are interesting examples of information organized through associative links, somewhat like semantic or associative networks. These systems are interactive window systems connected to a data base through associative links. Unlike normal text which is read in a linear fashion, hypertext can be browsed in a nonlinear way by moving through a network of information nodes which are linked bidirectionally through associative relationships. Users of hypertext systems can wander through the data base scanning text and graphics, creating new information nodes and linkages or modify existing ones. This approach to documentation use is said to more closely match the cognitive process. It provides a new approach to information access and organization for authors, researchers, and other users of large bodies of information.

11.4 MEMORY ORGANIZATION SYSTEMS

HAM, a Model of Memory

One of the earliest computer models of memory was the Human Associative Memory (HAM) system developed by John Anderson and Gordon Bower (1973). This memory is organized as a network of propositional binary trees. An example of a simple tree which represents the statement "In a park a hippie touched a debutante" is illustrated in Figure 11.6. When an *informant* asserts this statement to HAM, the system parses the sentence and builds a binary tree representation. Nodes in the tree are assigned unique numbers, while links are labeled with the following functions:

C: context for tree fact
e: set membership
F: a fact
L: a location
O: object
P: predicate
R: relation
S: subject
T: time

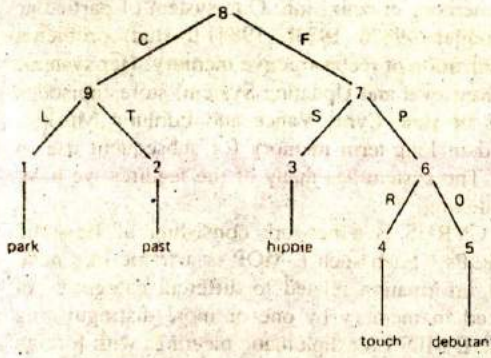
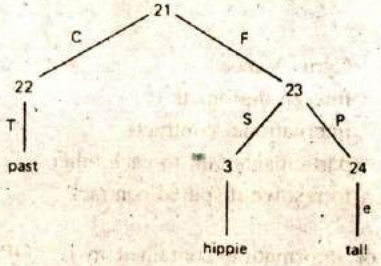


Figure 11.6 Organization of knowledge in HAM.

As HAM is informed of new sentences, they are parsed and formed into new tree-like memory structures or integrated with existing ones. For example, to add the fact that the hippie was tall, the following subtree is attached to the tree structure of Figure 11.6 by merging the common node hippie (node 3) into a single node.



When HAM is posed with a query, it is formed into a tree structure called a probe. This structure is then matched against existing memory structures for the best match. The structure with the closest match is used to formulate an answer to the query.

Matching is accomplished by first locating the leaf nodes in memory that match leaf nodes in the probe. The corresponding links are then checked to see if they have the same labels and in the same order. The search process is constrained by searching only node groups that have the same relation links, based on recency of usage. The search is not exhaustive and nodes accessed infrequently may be forgotten. Access to nodes in HAM is accomplished through word indexing in LISP (node words in tree structures are accessed directly through property lists or A-lists).

Memory Organization with E-MOPs

Roger Schank and his students at Yale University have developed several computer systems which perform different functions related to the use of natural language

text, knowledge representation, and memory organization. One system of particular interest was developed by Janet Kolodner (1983a, 1983b, 1984) to study problems associated with the retrieval and organization of reconstructive memory. Her system, called CYRUS (Computerized Yale Retrieval and Updating System) stores episodes from the lives of former secretaries of state Cyrus Vance and Edmund Muskie. The episodes are indexed and stored in long-term memory for subsequent use in answering queries posed in English. The system has many of the features we have described above under conceptual indexing.

The basic memory model in CYRUS is a network consisting of Episodic Memory Organization Packets (E-MOPs). Each such E-MOP is a frame-like node structure which contains conceptual information related to different categories of episodic events. E-MOPs are indexed in memory by one or more distinguishing features. For example, there are basic E-MOPs for diplomatic meetings with foreign dignitaries, specialized political conferences, traveling, sightseeing, negotiations, state dinners, as well as other basic events related to diplomatic state functions. The diplomatic-meeting E-MOP, called $MEET, contains information which is common to all diplomatic meeting events. The common information which characterizes such an E-MOP is called its content. For example, $MEET might contain the following information:

actor	:	Cyrus Vance
participants	:	foreign diplomats
topics	:	international contracts
actions	:	participants talk to each other
goals	:	to resolve disputed contract

A second type of information contained in E-MOPs are the indices which index either individual episodes or other E-MOPs which have become specializations of their parent E-MOPs. For instance, specific diplomatic meetings are indexed by features unique to the individual meetings such as location, actual topic discussed, or the actual meeting participants. A typical $MEET E-MOP which has indices to two particular event meetings EV1 and EV2, is illustrated in Figure 11.7.

One of the meetings indexed was between Vance and Gromyko of the USSR in which they discussed SALT (arms limit talks). This is labeled as event EV1 in the figure. The second meeting was between Vance and Begin of Israel in which they discussed Arab-Israeli peace. This is labeled as event EV2. Note that each of these events can be accessed through more than one feature (index). For example, EV1 can be located from the $MEET event through a topic value of "Arab-Israel peace," through a participants' nationality value of "Israel," through a participants' occupation value of "head of state," and so on.

As new diplomatic meetings are entered into the system, they are either integrated with the $MEET E-MOP as a separately indexed event or merged with another event to form a new specialized meeting E-MOP. When several events belonging

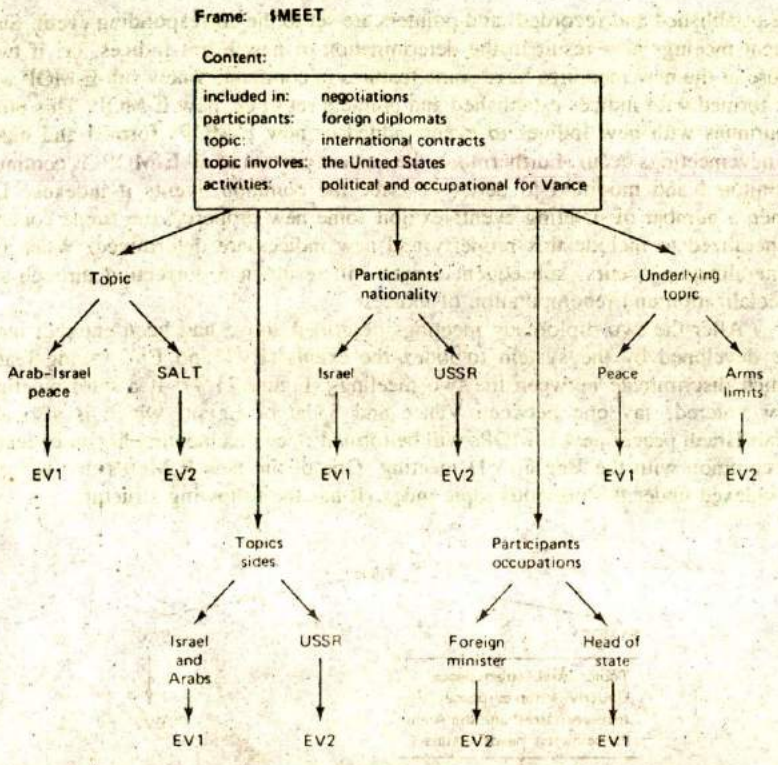


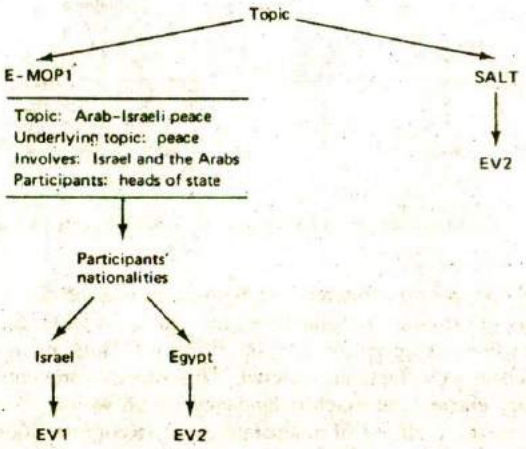
Figure 11.7 An example of an EMOP with two indexed events EV1 and EV2

to the same MOP category are entered, common event features are used to generalize the E-MOP. This information is collected in the frame contents. Specialization may also be required when over-generalization has occurred. Thus, memory is continually being reorganized as new facts are entered. This process prevents the addition of excessive memory entries and much redundancy which would result if every event entered resulted in the addition of a separate event. Reorganization can also cause forgetting, since originally assigned indices may be changed when new structures are formed. When this occurs, an item cannot be located; so the system attempts to derive new indices from the context and through other indices by reconstructing related events.

To see how CYRUS builds and maintains a memory organization, we briefly examine how a basic E-MOP grows and undergoes revision with time. Initially, the $MEET E-MOP of Figure 11.7 would consist of the content part of the frame only. Then, after a first meeting occurred, indices relevant and unique to that meeting

are established and recorded, and pointers are set to the corresponding event. Subsequent meeings also result in the determination of new event indices, or, if two or more of the new meetings have some features in common, a new sub-E-MOP would be formed with indices established and pointers set to the new E-MOP. This process continues with new indices to events added or new E-MOPs formed and indexed as new meetings occur. Furthermore, the content portion of all E-MOPs is continually monitored and modified to better describe the common events it indexes. Thus, when a number of meeting events exhibit some new property, the frame content is generalized to include this property and new indices are determined. When overgeneralization occurs, subsequent events will result in a correction through some specialization and recomputation of indices.

After the two diplomatic meetings described above had been entered, indices are developed by the system to index the events (EV1 and EV2) using features which discriminate between the two meetings (Figure 11.7). If a third meeting is now entered, say one between Vance and Sadat of Egypt, which is also about Arab-Israeli peace, new E-MOPs will be formed since this meeting has some features in common with the Begin (V1) meeting. One of the new E-MOPs that is formed is indexed under the previous topic index. It has the following structure:



The key issues in this type of organization are the same as those noted earlier. They are (1) the selection and computation of good indices for new events so that similar events can be located in memory for new event integration, (2) monitoring and reorganization of memory to accommodate new events as they occur, and (3) access of the correct event information when provided clues for retrieval.

11.5 SUMMARY

Effective memory organization can facilitate remembering (knowledge retrieval and access), memory management (knowledge reorganization), and forgetting. These are important components of the overall knowledge manipulation process. They are essential adjuncts to the reasoning and learning functions.

Popular forms of intelligent knowledge organization schemes are associative networks, networks of frames, or other structured representation schemes. These schemes will permit the effective organization and revision of knowledge into meaningful categories. Items within each category should share common features and individual items are best indexed by their differences. Access to categories and items can then be made through indices determined by content rather than meaningless keys. The best type of indices are those with good predictive power. Thus, relevant features determined by content and uniqueness to an item are good sources for index determination.

The CYRUS system is a good example of "intelligent" memory organization. It exhibits many of the desirable features possessed by human memories as described in this chapter. They include integrating and indexing events by context, memory reorganization by generalization or specialization, and the formation of new memory structures. Reorganization is performed to reflect commonalities and unique differences among events.

E-MOPs are the basic organization unit within CYRUS. They organize events by indexing them according to their differences. Similarities between events make up the generalized episode descriptions, and separate E-MOPs are associated by causal links.

EXERCISES

11.1. What important characteristics should a computer memory organization system possess?

11.2. Explain why each of the characteristics named in Problem 11.1 are important.

11.3. What basic operations must a program perform in order to access specific chunks of knowledge?

11.4. Suppose 64-byte records are stored in blocks of size 2^8 bytes. Describe a suitable index file to access the records using the following keys (start with block address 1000).

rabbit	dog	cat	duck
chicken	pig	cow	rat
horse	ox	mule	parrot
gopher	mouse	deer	elk

11.5. If ten chunks of knowledge were each stored in records of 64 bytes and the records randomly stored in eight blocks of secondary memory, what would be the access

time when a block can be located and read on the average within 60 ms. and the time to search each record is one ms. per block? Compare this time to the time required to search a single block for the same information.

11.6. Referring to Problem 11.4, describe how a hashing method could be applied to search for the indicated records.

11.7. Draw a conceptual indexing tree structure using the same keys as those given in Problem 11.4, but with the addition of a generalized node named farm-animals.

11.8. Using the same label links as those used in HAM, develop propositional trees for the following sentences.
The birds were singing in the park.
John and Mary went dancing at the prom.
Do not drink the water.

11.9. For the previous problem, add the sentence "There are lots of birds and they are small and yellow."

11.10. Develop an E-MOP for a general episode to fill up a car with gasoline using the elements Actor, Participant, Objects, Actions, and Goals.

11.11. Show how the E-MOP of Problem 11.10 would be indexed and accessed for the two events of filling the car at a self-service and at a full-service location.

11.12. Are the events of Problem 11.11 good candidates for specialized E-MOPs? Explain your answer.

11.13. Give an example of a hashing function that does not distribute key values uniformly over the key space.

11.14. Draw a small hypertext network that you might want to browse where the general network subject of artificial intelligence is used. Make up your own subtopics and show all linkages which you feel are useful, including link directions between subtopics.

11.15. Show how the E-MOP of Figure 11.7 would be generalized when peace was one of the topics discussed at every meeting.

11.16. Modify the E-MOP of Figure 11.7 to accommodate a new meeting between Vance and King Hussain of Jordan. The topic of their meeting is Palestinian refugees.