



Search: An Overview

Anne v.d.L. Gardner
Department of Computer Science
Stanford University
Stanford, California 94305

This article is the second planned excerpt from the Handbook of Artificial Intelligence being compiled at Stanford University. This overview of the Handbook chapter on search, like the overview of natural language research we printed in the first issue, introduces the important ideas and techniques, which are discussed in detail later in the chapter. Cross-references to other articles in the Handbook have been removed -- terms discussed in more detail elsewhere are italicized. The author would like to note that this article draws on material generously made available by Nils Nilsson for use in the Handbook.

In Artificial Intelligence, the terms *problem solving* and *search* refer to a large body of core ideas that deal with deduction, inference, planning, commonsense reasoning, theorem proving, and related processes. Applications of these general ideas are found in programs for natural language understanding, information retrieval, automatic programming, robotics, scene analysis, game playing, expert systems, and mathematical theorem proving. In this chapter of the Handbook we examine search as a tool for problem solving in a more limited area. Most of the examples considered are problems that are relatively easy to formalize. Some typical problems are:

1. finding the solution to a puzzle,
2. finding a proof for a theorem in logic or mathematics,

3. finding the shortest path connecting a set of nonequidistant points (the traveling-salesman problem),
4. finding a sequence of moves that will win a game, or the best move to make at a given point in a game,
5. finding a sequence of transformations that will solve a symbolic integration problem.

This overview takes a general look at search in problem solving, indicating some connections with topics considered in other Handbook chapters. The remainder of the Search chapter is divided into three sections. The first describes the problem representations that form the basis of search techniques: *state-space representations*, *problem-reduction representations*, and *game trees*. The second section considers algorithms that use these representations. *Blind search* algorithms, which treat the search space syntactically, are contrasted with *heuristic* methods, which use information about the nature and structure of the problem domain to limit the search. Finally, the chapter reviews several well-known early programs based on search, together with some related *planning* programs.

Components of Search Systems

Problem-solving systems can usually be described in

terms of three main components. The first of these is a *database*, which describes both the current task-domain situation and the goal. The database can consist of a variety of different kinds of data structures including arrays, lists, sets of predicate calculus expressions, property list structures, and semantic networks. In a domain for automated theorem proving, for example, the current task-domain situation consists of assertions representing axioms, lemmas, and theorems already proved; the goal is an assertion representing the theorem to be proved. In information-retrieval applications, the current situation consists of a set of facts, and the goal is the query to be answered. In robot problem-solving, a current situation is a *world model* consisting of statements describing the physical surroundings of the robot, and the goal is a description that is to be made true by a sequence of robot actions.

The second component of problem-solving systems is a set of *operators* that are used to manipulate the database. Some examples of operators include:

1. in theorem proving, rules of inference such as modus ponens and resolution;
2. in chess, rules for moving chessmen;
3. in symbolic integration, rules for simplifying the forms to be integrated, such as integration by parts or trigonometric substitution.

Sometimes the set of operators consists of only a few general rules of inference that generate new assertions from existing ones. Usually it is more efficient to use a large number of very specialized operators that generate new assertions only from very specific existing ones.

The third component of a problem-solving system is a *control strategy* for deciding what to do next--in particular, what operator to apply and where to apply it. Sometimes control is highly centralized, in a separate control executive that decides how problem-solving resources should be expended. Sometimes control is diffusely spread among the operators themselves.

The choice of a control strategy affects the contents and organization of the database. In general, the object is to achieve the goal by applying an appropriate sequence of operators to an initial task-domain situation. Each application of an operator modifies the situation in some way. If several different operator sequences are worth considering, the representation often maintains data structures showing the effects on the task situation of each alternative sequence. Such a representation permits a control strategy that investigates various operator sequences in parallel or that alternates attention among a number of sequences that look relatively promising. Algorithms of this sort assume a database containing descriptions of multiple task-domain situations or *states*. It

may be, however, that the description of a task-domain situation is too large for multiple versions to be stored explicitly; in this case, a *backtracking* control strategy may be used. A third approach is possible in some types of problems such as theorem proving, where the application of operators can add new assertions to the description of the task-domain situation but never can require the deletion of existing assertions. In this case, the database can describe a single, incrementally changing task-domain situation; multiple or alternative descriptions are unnecessary.

Reasoning Forward and Reasoning Backward

The application of operators to those structures in the database that describe the task-domain situation--to produce a modified situation--is often called *reasoning forward*. The object is to bring the situation, or problem state, forward from its initial configuration to one satisfying a goal condition. For example, an initial situation might be the placement of chessmen on the board at the beginning of the game; the desired goal, any board configuration that is a checkmate; and the operators, rules for the legal moves in chess.

An alternative strategy, *reasoning backward*, involves another type of operator, which is applied, not to a current task-domain situation, but to the goal. The goal statement, or problem statement, is converted to one or more subgoals that are (one hopes) easier to solve and whose solutions are sufficient to solve the original problem. These subgoals may in turn be reduced to sub-subgoals, and so on, until each of them is accepted to be a trivial problem or its subproblems have been solved. For example, given an initial goal of integrating $1/(\cos^2 x)$ dx , and an operator permitting $1/(\cos x)$ to be rewritten as $\sec x$, one can work backward toward a restatement of the goal in a form whose solution is immediate: The integral of $\sec^2 x$ is $\tan x$.

The former approach is said to use *forward reasoning* and to be *data-driven* or *bottom-up*. The latter uses *backward reasoning* and is *goal-directed* or *top-down*. The distinction between forward and backward reasoning assumes that the current task-domain situation or state is distinct from the goal. If one chooses to say that a current state is the state of having a particular goal, the distinction naturally vanishes.

Much human problem-solving behavior is observed to involve reasoning backward, and many artificial intelligence programs are based on this general strategy. In addition, combinations of forward and backward reasoning are possible. One important AI technique involving forward and backward reasoning is called *means-ends analysis*; it involves comparing the current goal

with a current task-domain situation to extract a *difference* between them. This difference is then used to index the (forward) operator most relevant to reducing the difference. If this especially relevant operator cannot be immediately applied to the present problem state, subgoals are set up to change the problem state so that the relevant operator can be applied. After these subgoals are solved, the relevant operator is applied and the resulting, modified situation becomes a new starting point from which to solve for the original goal.

State Spaces and Problem Reduction

A problem-solving system that uses forward reasoning and whose operators each work by producing a single new object--a new state--in the database is said to represent problems in a *state-space representation*.

For backward reasoning, a distinction may be drawn between two cases. In one, each application of an operator to a problem yields exactly one new problem, whose size or difficulty is typically slightly less than that of the previous problem. Systems of this kind are also referred to, in this chapter, as employing state-space representations. Two instances of such representations, described in other articles, are the Logic Theorist program (Newell, Shaw, and Simon, 1963) and the backward-reasoning part of *bidirectional search* (Pohl, 1971).

A more complex kind of backward reasoning occurs if applying an operator may divide the problem into a set of subproblems, perhaps each significantly smaller than the original. An example would be an operator changing the problem of integrating $2/(x^2 - 1) dx$ into the three subproblems of integrating $1/(x - 1) dx$, integrating $-1/(x + 1) dx$, and adding the results. A system using this kind of backward reasoning, distinguished by the fact that its operators can change a single object into a conjunction of objects, will be said to employ a *problem-reduction representation*.

There may or may not be constraints on the order in which the subproblems generated by a problem-reduction system can be solved. Suppose, for example, that the original problem is to integrate $(f(x) + g(x) dx)$. Applying the obvious operator changes it to the new problem consisting of two integrations, $f(x) dx$ and $g(x) dx$. Depending on the representation, the new problem can be viewed as made up of either (a) two integration subproblems that can be solved in any order or (b) two integration subproblems plus the third subproblem of summing the integrals. In the latter case, the third task cannot be done until the first two have been completed.

Besides the state-space and problem-reduction approaches, other variations on problem representation are

possible. One is used in game-playing problems, which differ from most other problems by virtue of the presence of adversary moves. A game-playing problem must be represented in a way that takes into account the opponent's possible moves as well as the player's own. The usual representation is a *game tree*, which shares many features of a problem-reduction representation. Detailed examples of game-tree representations, as well as of state-space and problem-reduction representations, are given later in the chapter. Examples may also be found in Nilsson's texts.

Another variation is relevant to theorem-proving systems, many of which use forward reasoning and operators (rules of inference) that act on conjunctions of objects in the database. Although the representations discussed here assume that each operator takes only a single object as input, it is possible to define a *theorem-proving representation* that provides for multiple-input, single-output operators (see Kowalski, 1972).

Graph Representation

In either a state-space or a problem-reduction representation, achieving the desired goal can be equated with finding an appropriate finite sequence of applications of available operators. While what one is primarily interested in--the goal situation or the sequence that leads to it--may depend on the problem, the term *search* can always be understood, without misleading consequences, as referring to the search for an appropriate operator sequence.

Tree structures are commonly used in implementing control strategies for the search. In a state-space representation, a tree may be used to represent the set of problem states produced by operator applications. In such a representation, the root node of the tree represents the initial problem situation or state. Each of the new states that can be produced from this initial state by the application of just one operator is represented by a *successor node* of the root node. Subsequent operator applications produce successors of these nodes, and so on. Each operator application is represented by a directed *arc* of the tree. In general, the states are represented by a *graph* rather than by a tree, since there may be different paths from the root to any given node. Trees are an important special case, however, and it is usually easier to explain their use than that of graphs.

Besides these ordinary trees and graphs, which are used for state-space representations, there are also specialized ones called **AND/OR graphs** that are used with problem-solving methods involving problem reduction. For problems in which the goal can be reduced to sets of subgoals, **AND/OR graphs** provide a means for keeping track of which subgoals have been attempted and which

combinations of subgoals are sufficient to achieve the original goal.

The Search Space

The problem of producing a state that satisfies a goal condition can now be formulated as the problem of searching a graph to find a node whose associated state description satisfies the goal. Similarly, search based on a problem-reduction representation can be formulated as the search of an **AND/OR** graph.

It should be noted that there is a distinction between the graph to be searched and the tree or graph that is constructed as the search proceeds. In the latter, nodes and arcs can be represented by explicit data structures; the only nodes included are those for which paths from the initial state have actually been discovered. This explicit graph, which grows as the search proceeds, will be referred to as a *search graph* or *search tree*.

In contrast, the graph to be searched is ordinarily not explicit. It may be thought of as having one node for every state to which there is a path from the root. It may even be thought of, less commonly, as having one node for every state that can be described, whether or not a path to it exists. The implicit graph will be called the *state space* or, if generalized to cover non-state-space representations such as **AND/OR** graphs or game trees, the *search space*. Clearly, many problem domains (such as theorem proving) have an infinite search space, and the search space in others, though finite, is unimaginably large. Estimates of search-space size may be based on the total number of nodes (however defined) or on other measures. In chess, for example, the number of different complete plays of the average-length game has been estimated at 10^{120} (Shannon, 1950, 1956), although the number of "good" games is much smaller (see Good, 1968). Even for checkers, the size of the search space has been estimated at 10^{40} (Samuel, 1963).

Searching now becomes a problem of making just enough of the search space explicit in a search graph to contain a solution of the original goal. If the search space is a general graph, the search graph may be a subgraph, a subgraph that is also a tree, or a tree obtained by representing distinct paths to one search space node with duplicate search graph nodes.

Limiting Search

The critical problem of search is the amount of time and space necessary to find a solution. As the chess and checkers estimates suggest, exhaustive search is rarely feasible for nontrivial problems. Examining all sequences of n moves, for example, would require operating in a search space in which the number of nodes grows exponentially with n . Such a phenomenon is called a

combinatorial explosion.

There are several complementary approaches to reducing the number of nodes that a search must examine. One important way is to recast the problem so that the size of the search space is reduced. A dramatic, if well-known, example is the mutilated chessboard problem:

Suppose two diagonally opposite corner squares are removed from a standard 8 by 8 square chessboard. Can 31 rectangular dominoes, each the size of exactly two squares, be so placed as to cover precisely the remaining board? (Raphael, 1976, p. 31)

If states are defined to be configurations of dominoes on the mutilated board, and an operator has the effect of placing a domino, the search space for this problem is very large. If, however, one observes that every domino placed must cover both a red square and a black one and that the squares removed are both of one color, the answer is immediate. Unfortunately, little theory exists about how to find good problem representations. Some of the sorts of things such a theory would need to take into account are explored by Amarel (1968), who gives a sequence of six representations for a single problem, each reducing the search space size by redefining the states and operators.

A second aspect concerns search efficiency within a given search space. Several graph- and tree-searching methods have been developed, and these play an important role in the control of problem-solving processes. Of special interest are those graph-searching methods that use *heuristic knowledge* from the problem domain to help focus the search. In some types of problems, these *heuristic search* techniques can prevent a combinatorial explosion of possible solutions. Heuristic search is one of the key contributions of AI to efficient problem solving. Various theorems have been proved about the properties of search techniques, both those that do and those that do not use heuristic information. Briefly, it has been shown that certain types of search methods are guaranteed to find optimal solutions (when such exist). Some of these methods, under certain comparisons, have also been shown to find solutions with minimal search effort. Graph- and tree-searching algorithms, with and without the use of heuristic information, are discussed at length later in the chapter.

A third approach addresses the question: Given one representation of a search problem, can a problem-solving system be programmed to find a better representation automatically? The question differs from that of the first approach to limiting search in that here it is the program, not the program designer, that is asked to find the improved representation. One start on answering the question was made in the **STRIPS** program (Fikes and Nilsson, 1971; Fikes, Hart, and Nilsson, 1972). **STRIPS** augments its initial set of operators by discovering, generalizing, and remembering *macro-operators*, composed of sequences of primitive operators, as it gains

problem-solving experience. Another idea was used in the **ABSTRIPS** program (Sacerdoti, 1974), which implements the idea of *planning*, in the sense of defining and solving problems in a search space from which unimportant details have been omitted. The details of the solution are filled in (by smaller searches within the more detailed space) only after a satisfactory outline of a solution, or *plan*, has been found. Planning is a major topic itself, discussed in Volume III of the Handbook.

The Meaning of Heuristic and Heuristic Search

Although the term *heuristic* has long been a key word in AI, its meaning has varied both among authors and over time. A brief review of the ways *heuristic* and *heuristic search* have been used may provide a useful warning against taking any single definition too seriously.

As an adjective, the most frequently quoted dictionary definition for *heuristic* is "serving to discover." As a noun, referring to an obscure branch of philosophy, the word meant the study of the methods and rules of discovery and invention (see Polya, 1957, p. 112).

When the term came into use to describe AI techniques, some writers made a distinction between methods for discovering solutions and methods for producing them algorithmically. Thus, Newell, Shaw, and Simon stated in 1957: "A process that *may* solve a given problem, but offers no guarantees of doing so, is called a *heuristic* for that problem" (Newell, Shaw, and Simon, 1963, p. 114). But this meaning was not universally accepted. Minsky, for example, said in a 1961 paper:

The adjective "heuristic," as used here and widely in the literature, means *related to improving problem-solving performance*, as a noun it is also used in regard to any method or trick used to improve the efficiency of a problem-solving program. But imperfect methods are not necessarily heuristic, nor vice versa. Hence "heuristic" should not be regarded as opposite to "foolproof", this has caused some confusion in the literature (Minsky, 1963, p. 407n)

These two definitions refer, though vaguely, to two different sets: devices that improve efficiency and devices that are not guaranteed. Feigenbaum and Feldman (1963) apparently limit *heuristic* to devices with both properties:

A *heuristic* (*heuristic rule*, *heuristic method*) is a rule of thumb, strategy, trick, simplification, or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions; in fact, they do not guarantee any solution at all; *all that can be said for a useful heuristic is that it offers solutions which are good enough most of the time* (p. 6; italics in original)

Even this definition, however, does not always agree with common usage, because it lacks a historical dimension. A device originally introduced as a heuristic in Feigenbaum and Feldman's sense may later be shown to

guarantee an optimal solution after all. When this happens, the label *heuristic* may or may not be dropped. It has not been dropped, for example, with respect to the A^* algorithm. *Alpha-beta pruning*, on the other hand, is no longer called a heuristic. (For descriptions of both devices, see Nilsson.)

It should be noted that the definitions quoted above, ranging in time from 1957 to 1963, refer to heuristic rules, methods, and programs, but they do not use the term *heuristic search*. This composite term appears to have been first introduced in 1965 in a paper by Newell and Ernst, "The Search for Generality" (see Newell and Simon, 1972, p. 888). The paper presented a framework for comparing the methods used in problem-solving programs up to that time. The basic framework, there called heuristic search, was the one called *state-space search* in the present chapter. Blind search methods were included in the heuristic search paradigm.

A similar meaning for heuristic search appears in Newell and Simon (1972, pp. 91-105). Again, no contrast is drawn between heuristic search and blind search; rather, heuristic search is distinguished from a problem-solving method called *generate and test*. The difference between the two is that the latter simply generates elements of the search space (i.e., states) and tests each in turn until it finds one satisfying the goal condition; whereas in heuristic search the order of generation can depend both on information gained in previous tests and on the characteristics of the goal. But the Newell and Simon distinction is not a hard and fast one. By the time of their 1976 Turing Lecture, they seem to have collapsed the two methods into one:

Heuristic Search A second law of qualitative structure for AI is that symbol systems solve problems by generating potential solutions and testing them, that is, by searching (Newell and Simon, 1976, p. 126)

In the present chapter, the meaning attached to *heuristic search* stems not from Newell and Simon but from Nilsson, whose 1971 book provided the most detailed and influential treatment of the subject that had yet appeared (see also Nilsson, 1980). For Nilsson, the distinction between heuristic search and blind search is the important one. Blind search corresponds approximately to the systematic generation and testing of search-space elements, but it operates within a formalism that leaves room for additional information about the specific problem domain to be introduced, rather than excluding it by definition. If such information, going beyond that needed merely to formulate a class of problems as search problems, is in fact introduced, it may be possible to restrict search drastically. Whether or not the restriction is foolproof, the search is then called heuristic rather than blind. ■

(references continued page 23)

For Decsystem-20, Tops-20, Vax, and VMS:

Consult your friendly local DEC salesperson
Interlisp is available through DECUS.

For Interlisp on VAX:

Mel Pirtle
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90291
(PIRTLE @ ISIB)

For Maclisp and NIL:

Jon L. White
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Mass. 02139
(JONL @ MIT-MC)

For Eunice:

David Kashtan (technical questions)
Chuck Untulis (administrative questions)
SRI International
Computer Resources
333 Ravenswood Ave.
Menlo Park, California 94025
(KASHTAN @ SRI-KL, UNTULIS @ SRI-KL)

For Lisp Machines:

Russell Nofsker	Steve Wyle
Symbolics, Incorporated	Lisp Machines, Incorporated
605 Hightree Road	163 N. Mansfield Ave
Santa Monica, Ca. 90402	Los Angeles, Ca 90036
(213) 459-6040	(213) 938-8888

For PERQs:

Three Rivers Computer Corporation
720 Gross St.
Pittsburgh, Pa 15224
(412) 621-6250

For Jericho:

Jim Calvin
Bolt, Beranek, and Newman
50 Moulton St
Cambridge, Mass. 02138
(617) 491-1850 x4615
CALVIN BBN-TENEXG

For Spice:

Scott E. Fahlman
Department of Computer Science
Carnegie-Mellon University
Schenley Park
Pittsburgh, Pa 15213
(FAHLMAN @ CMUA)

Search

(continued from page 6)

References

- Amarel, S. On representations of problems of reasoning about actions. In D. Michie (Ed.), *Machine Intelligence 3*. New York: American Elsevier, 1968. Pp 131-171
- Feigenbaum, E. A., and Feldman, J. (Eds.) *Computers and Thought*. New York: McGraw-Hill, 1963
- Fikes, R. E., Hart, P., and Nilsson, N. J. Learning and executing generalized robot plans. *Artificial Intelligence*, 1972, 3, 251-288
- Fikes, R. E., and Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1971, 2, 189-208
- Good, I. J. A five-year plan for automatic chess. In E. Dale and D. Michie (Eds.), *Machine Intelligence 2*. New York: American Elsevier, 1968. Pp 89-118
- Kowalski, R. And-or graphs, theorem-proving graphs, and bi-directional search. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 7*. New York: Wiley, 1972. Pp 167-194
- Minsky, M. Steps toward artificial intelligence. In Feigenbaum and Feldman, 1963. Pp 406-450
- Newell, A., and Ernst, G. The search for generality. In W. A. Kalenich (Ed.), *Information Processing 1965: Proc IFIP Congress 65*. Washington: Spartan Books, 1965. Pp 17-24
- Newell, A., Shaw, J. C., and Simon, H. A. Empirical explorations with the logic theory machine: A case history in heuristics. In Feigenbaum and Feldman, 1963. Pp 109-133
- Newell, A., and Simon, H. A. *Human Problem Solving*. Englewood Cliffs, N.J.: Prentice-Hall, 1972
- Newell, A., and Simon, H. A. Computer science as empirical inquiry: Symbols and search. The 1976 ACM Turing Lecture. *Comm ACM*, 1976, 19, 113-126
- Nilsson, N. J. *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971
- Nilsson, N. J. *Principles of Artificial Intelligence*. Palo Alto, Calif: Tioga, 1980
- Pohl, I. Bi-directional search. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 6*. New York: American Elsevier, 1971. Pp 127-140
- Polya, G. *How to Solve It* (2nd ed.). New York: Doubleday Anchor, 1957
- Raphael, B. *The Thinking Computer*. San Francisco: Freeman, 1976
- Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 1974, 5, 115-135
- Samuel, A. L. Some studies in machine learning using the game of checkers. In Feigenbaum and Feldman, 1963. Pp 71-105
- Shannon, C. E. Programming a computer for playing chess. *Philosophical Magazine* (Series 7), 1950, 41, 256-275
- Shannon, C. E. A chess-playing machine. In J. R. Newman (Ed.), *The World of Mathematics* (Vol 4). New York: Simon and Schuster, 1956. Pp 2124-2133