The field of linguistics shares an interest in the theory of grammars and languages with AI. Both fields have a desire to build a well-founded theory, and to see the development of systems that understand natural languages, that can synthesize speech, and that are capable of language translations.

Finally, AI has some overlap with almost all fields in that it offers the potential for broad applications. Applications have already been proven in such areas as medicine, law, manufacturing, economics, banking, biology, chemistry, defense, civil engineering, and aerospace to name a few.

Q.5. Define AI. Outline the nature of problems that led to the development of AI. Explain typical Task Domains of AI.

[S-98, S-03, W-03]

OR Explain Mundane, formula and task domains in AI with suitable examples. ***[S-02]***

Ans. Much of the early work in the field focused on formal tasks, such as game playing and theorem proving. Samuel wrote a checkers-playing program that not only played games with opponents but also used its experience at those games to improve its later performance. Chess also received a good deal of attention. The Logic Theorist was an early attempt to prove mathematical theorems. Gelernter's theorem prover explored another area of mathematics : geometry. Game playing and theorem proving share the property that people who do them well are considered to be displaying intelligence. Despite this, it appeared initially that computers could perform well at those tasks simply by being fast at exploring a large number of solution paths and then selecting the best one. It was thought that this process required very little knowledge and could therefore be programmed easily. This assumption turned out to be false since no computer is fast enough to overcome the combinatorial explosion generated by most problems.

Another early foray into AI focused on the sort of problem solving that we do everyday when we decide how to get to work in

the morning, often called commonsense reasoning. It includes reasoning about physical objects and their relationships to each other (i.e., an object can be in only one place at a time), as well as reasoning about actions and their consequences (e.g., if you let go of something, it will fall to the floor and maybe break). To investigate this sort of reasoning, Newell, Shaw, and Simon built the General Problem Solver (GPS), which they applied to several commonsense tasks as well as to the problem of performing symbolic manipulations of logical expressions. Again, no attempt was made to create a program with a large amount of knowledge about a particular problem domain. Only quite simple tasks were selected.

As AI research progressed and techniques for handling larger amounts of world knowledge were developed, some progress was made on the tasks just described and new tasks could reasonably be attempted. These include perception (vision and speech), natural language understanding, and problem solving in specialized domains such as medical diagnosis and chemical analysis.

Perception of the world around us is crucial to our survival. Animals with much less intelligence than people are capable of more sophisticated visual perception than are current machines. Perceptual tasks are difficult because they involve analog (rather than digital) signals; the signals are typically very noisy and usually a large number of things (some of which may be partially obscuring other) must be perceived at once.

The ability to use language to communicate a wide variety of ideas is perhaps the most important thing that separates humans from the other animals. The problem of understanding spoken language is a perceptual problem. But suppose we simplify the problem by restricting it to written language. This problem, usually referred to as natural language understanding, is still extremely difficult. In order to understand sentences about a topic, it is necessary to know

not only a lot about the language itself (its vocabulary and grammar) but also a good deal about the topic so that unstated assumptions can be recognized.

In addition to these mundane tasks, many people can also perform one or may be more specialized tasks in which carefully acquired expertise is necessary. Examples of such tasks include engineering design, scientific discovery, medical diagnosis, and financial planning. Programs that can solve problems in these domains also fall under the aegis of artificial intelligence. Figure lists some of the tasks that are the targets of work in AI.

A person who knows how to perform tasks from several of the categories shown in the figure learns the necessary skills in a standard order. First perceptual, linguistic, and commonsense skills are learned. Later (and of course for some people, never) expert skills such as engineering, medicine, or finance are acquired. It might seem to make sense then that the earlier skills are easier and thus more amenable to computerized duplication than are the later, more specialized ones. For this reason, much of the initial AI work was concentrated in those early areas. But it turns out that this naïve assumption is not right. Although expert skills require knowledge that many of us do not have, they often require much less knowledge and deal inside the program.

As a result, the problem areas where AI is now flourishing most as a practical discipline (as opposed to a purely research one) are primarily the domains that require only specialized expertise without the assistance of commonsense knowledge. There are now thousands of programs called expert systems in day-to-day operation throughout all areas of industry and government. Each of these systems attempts to solve part, or perhaps all, of a practical, significant problem that previously required scarce human expertise.

Mundane Tasks

- Perception
  - Vision
  - Speech
- Natural language
  - Understanding
  - Generation
  - Translation
- Commonsense reasoning
- Robot control

Formal Tasks

- Games
  - Chess
  - Backgammon
  - Checkers
  - Go
- Mathematics
  - Geometry
  - Logic
  - Integral calculus
  - Proving properties of programs

Expert Tasks

- Engineering
  - Design
  - Fault finding
  - Manufacturing planning
- Scientific analysis
- Medical diagnosis
- Financial analysis

Fig. Some of the Task Domains of Artificial Intelligence

Q.6. Define physical symbol system and it's hypothesis.

Ans. A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). Thus, a symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way (such as one token being next to another). At any instant of time the system will contain a collection of these symbol structures. Besides these

structures, the system also contains a collection of processes that operate on expressions to produce other expressions : processes of creation, modification, reproduction and destruction. A physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbolic expressions themselves.

Then state the hypothesis as

The Physical Symbol System Hypothesis : A physical symbol system has the necessary and sufficient means for general intelligent action.

Q.7. What is an AI technique? *[W-97, W-98, S-02]*

OR Construct a script for shopping in a supermarket. What are different constituents of a script? *[W-02]*

Ans. Artificial intelligence problems span a very broad spectrum. They appear to have very little in common except that they are hard.

One of the few hard and fast results to come out of the first three decades of AI research is that intelligence requires knowledge. To compensate for its one overpowering asset, indispensability, knowledge possesses some less desirable properties, including :

- It is voluminous.
- It is hard to characterize accurately.
- It is constantly changing.
- If differs from data by being organized in a way that corresponds to the ways it will be used.

We are forced to conclude that an AI technique is a method that exploits knowledge that should be represented in such a way that :

- The knowledge captures generalizations. In other words, it is not necessary to represent separately each individual situation. Instead, situations that share important properties are grouped together. If knowledge does not have this property, inordinate amounts of memory and updating will be required. So we usually call something without this property "data" rather than knowledge.

- It can be understood by people who must provide it. Although for many programs, the bulk of the data can be acquired automatically (for example, by taking readings from a variety of instruments), in many AI domains, most of the knowledge a program has must ultimately be provided by people in terms they understand.
- It can easily be modified to correct errors and to reflect changes in the world and in our world view.
- It can be used in a great many situations even if it is not totally accurate or complete.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must usually be considered.

Although AI techniques must be designed in keeping with these constraints imposed by AI problems, there is some degree of independence between problems and problem solving techniques. It is possible to solve AI problems without using AI techniques. And it is possible to apply AI techniques to the solution of non-AI problems. This is likely to be a good thing to do for problems that possess many of the same characteristics as do AI problems. In order to try to characterize AI techniques in as problem-independent a way as possible, there are very different problems and a series of approaches for solving each of them.

1. Tic-Tac-Toe :-

In this section, we present a series of three programs to play tic-tac-toe. The programs in this series increase in :

- Their complexity
- Their use of generalizations
- The clarity of their knowledge
- The extensibility of their approach

Thus they move toward being representations of what we call

AI techniques.

Program 1

Data Structures :

Board A nine-element vector representing the board, where the elements of the vector correspond to the board positions as follows :

1	2	3
4	5	6
7	8	9

An element contains the value 0 if the corresponding square is blank, 1 if it is filled with an X, or 2 if it is filled with an O.

Movetable A large vector of 19,683 elements (3^9), each element of which is a nine-element vector. The contents of this vector are chosen specifically to allow the algorithm to work.

The Algorithm :

To make a move, do the following :

1. View the vector Board as a ternary (base three) number. Convert it to a decimal number.
2. Use the number computed in step 1 as an index into Movetable and access the vector stored there.
3. The vector selected in step 2 represents the way the board will look after the move that should be made. So set Board equal to that vector.

Comments :

This program is very efficient in terms of time. And, in theory, it could play an optimal game of tic-tac-toe. But it has several disadvantages :

- It takes a lot of space to store the table that specifies the correct move to make from each board position.
- Someone will have to do a lot of work specifying all the entries in the movetable.
- It is very unlikely that all the required movetable entries can be determined and entered without any errors.

- If we want to extend the game, say to three dimensions, we would have to start from scratch, and in fact this technique would no longer work at all, since 3^{27} board positions would have to be stored, thus overwhelming present computer memories.

The technique embodied in this program does not appear to meet any of our requirements for a good AI technique. Let's see if we can do better.

Program 2

Data Structures :

Board A nine-element vector representing the board, as described for Program 1. But instead of using the numbers 0.1 or 2 in each element, we store 2 (indicating bank), 3 (indicating X,) or 5 (indicating O).

Turn An integer indicating which move of the game is about to be played; 1 indicates the first move, 9 the last.

The Algorithm :

The main algorithm uses three subprocedures :

Make 2 Returns 5 if the center square of the board is blank, that is, if Board [5] = 2. Otherwise, this function returns any blank noncorner square (2, 4, 6, or 8).

Posswin(p) Returns 0 if player p cannot win on his next move; otherwise, it returns the number of the square that constitutes a winning move. This function will enable the program both to win and to block the opponent's win. Posswin operates by checking, one at a time, each of the rows, columns, and diagonals. Because of the way values are numbered, it can test on entire row (column or diagonal) to see if it is a possible win by multiplying the values of its squares together. If the product is 18 ($3 \times 3 \times 2$), then X can win. If the product is 50 ($5 \times 5 \times 2$), then O can win. If we find a winning row, we determine which element is blank, and return the number of that square.

Go(n) Makes a move in square n. This procedure sets

Board[n] to 3 if Turn is odd, or 5 if Turn is even. It also increments Turn by one.

The algorithm has a built-in strategy for each move it may have to make. It makes the odd-numbered moves if it is playing X, the even-numbered moves if it is playing O. The strategy for each turn is as follows :

Turn = 1	Go(1) (upper left corner).
Turn = 2	If Board[5] is blank, Go(5), else Go(1).
Turn = 3	If Board [9] is blank, Go(9), else Go(3).
Turn = 4	If Posswin(X) is not 0 then Go(Posswin(X)) [i.e., block opponent's win], else Go(Make2).
Turn = 5	f Posswin(X) is not 0 then Go(Posswin(X) [i.e., win] else if Posswin(O) is not 0, then Go(Posswin(O)) [i.e., block win], else if Board[7] is blank, then Go(7),else Go(3). [Here the program is trying to make a fork.]
Turn = 6	If Posswin(O) is not 0 then Go(Posswin(O)), else if Posswin(X) is not 0, then Go(Posswin(X)), else Go(Make2).
Turn = 7	If Posswin(X) is not 0 then Go(Posswin(X)), else if Posswin(O) is not 0, then Go(Posswin(O)), else go anywhere that is blank.
Turn = 8	If Posswin(O) is not 0 then Go(Posswin(O)), else if Posswin(X) is not 0, then Go(Posswin(X)), else go anywhere that is blank.
Turn = 9	Same as Turn = 7.

Comments :

This program is not quite as efficient in terms of time as the first one since it has to check several conditions before making each move. But it is a lot more efficient in terms of space. It is also a lot easier to understand the program's strategy or to change the strategy if desired. But the total strategy has still been figured out in advance by the programmer. Any bugs in the programmer's tic-tac-toe playing skill will show up in the program's play. And we still cannot

generalize any of the program's knowledge to a different domain, such as three-dimensional tic-tac-toe.

Program 2′

This program is identical to Program 2 except for one change in the representation of the board. We again represent the board as a nine-element vector, but this time we assign board positions to vector elements as follows :

8	3	4
1	5	9
6	7	2

Notice that this numbering of the board produces a magic square : all the rows, columns, and diagonals sum to 15. This means that we can simplify the process of checking for a possible win. In addition to marking the board as moves are made, we keep a list, for each player, of the squares in which he or she has played. To check for a possible win for one player, we consider each pair of squares owned by that player and compute the difference between 15 and the sum of the two squares. If this difference is not positive or if it is greater than 9, then the original two squares were not collinear and so can be ignored. Otherwise, if the square representing the difference is blank, a move there will produce a win. Since no player can have more than four squares at a time, there will be many fewer squares examined using this scheme than there were using the more straightforward approach of Program 2. This shows how the choice of representation can have a major impact on the efficiency of a problem-solving program.

Comments :

This comparison raises an interesting question about the relationship between the way people solve problems and the way computers do. Why do people find the row-scan approach easier while the number-counting approach is more efficient for a computer? We do not know enough about how people work to answer that question completely. One part of the answer is that

people are parallel processors and can look at several parts of the board at once, whereas the conventional computer must look at the squares one at a time. Sometimes an investigation of how people solve problems sheds great light on how computers should do so. At other times, the differences in the hardware of the two seem so great that different strategies seem best. As we learn more about problem solving both by people and by machines, we may know better whether the same representations and algorithms are best for both people and machines.

Program 3

Data Structures :

Board Position A structure containing a nine-element vector representing the board, a list of board positions that could result from the next move, and a number representing an estimate of how likely the board position is to lead to an ultimate win for the player to move.

The Algorithm :

To decide on the next move, look ahead at the board positions that result from each possible move. Decide which position is best (as described below), make the move that leads to that position, and assign the rating of that best move to the current position.

To decide which of a set of board positions is best, do the following for each of them :

1. See if it is a win. If so, call it the best by giving it the highest possible rating.

2. Otherwise, consider all the moves the opponent could make next. See which of them is worst for us (by recursively calling this procedure). Assume the opponent will make that move. Whatever rating that move has, assign it to the node we are considering.

3. The best node is then the one with the highest rating.

This algorithm will look ahead at various sequences of moves in order to find a sequence that leads to a win. It attempts to maximize the likelihood of winning, while assuming that the opponent will try to minimize that likelihood. This algorithm is called the minimax procedure.

Comments :

This program will require much more time than either of the others since it must search a tree representing all possible move sequences before making each move. But it is superior to the other programs in one very big way. It could be extended to handle games more complicated than tic-tac-toe, for which the exhaustive enumeration approach of the other programs would completely fall apart. It can also be augmented by a variety of specific kinds of knowledge about games and how to play them.

Program 3 is an example of the use of an AI technique. For very small problems, it is less efficient than a variety of more direct methods. However, it can be used in situations where those methods would fail.

2. Question Answering :-

There is a series of programs that read in English text and then answer questions, also stated in English, about that text. This task differs from the last one in that it is more difficult now to state formally and precisely what our problem is and what constitutes correct solutions to it. For example, suppose that the input text were just the single sentence

Russia massed troops on the Czech border.

Then either of the following question-answering dialogues might occur.

Dialogue 1

Q : Why did Russia do this?

A : Because Russia thought that it could take political control of Czechoslovakia by sending troops.

Q : What should the United States do?

A : The United States should intervene militarily.

Q : Why did Russia do this?

A : Because Russia wanted to increase its political influence over Czechoslovakia.

Q : What should the United States do?

A : The United States should denounce the Russian action in the United Nations.

In the POLITICS program, answers were constructed by considering both the input text and a separate model of the beliefs and actions of various political entities, including Russia. When the model is changed, as it was between these two dialogues, the system's answers also change. In this example, the first dialogue was produced when POLITICS was given a model that was intended to correspond to the beliefs of a typical American conservative. The second dialogue occurred when POLITICS was given a model that was intended to correspond to the beliefs of a typical American liberal.

The general point here is that defining what it means to produce a correct answer to a question may be very hard.

In order to be able to compare the three programs, we illustrate all of them using the following text :

Mary went shopping for a new coat. She found a red one she really liked. When she got it home, she discovered that it went perfectly with her favorite dress.

We will also attempt to answer each of the following questions with each program :

Q.1 : What did Mary go shopping for?

Q.2: What did Mary find that she liked?

Q.3: Did Mary buy anything?

Program 1

This program attempts to answer questions using the literal input text. It simply matches text fragments in the questions against the input test.

Data Structures :

Question Patterns A set of templates that match common question forms and produce patterns to be used to match against inputs. Templates and patterns (which we call text patterns) are paired so that if a template matches successfully against an input question then its associated text patterns are used to try to find appropriate answers in the text. For example, if the template "Who

did x y" matches an input question, then the text pattern "x y z" is matched against the input text and the value of z is given as the answer to the question.

Text The input text stored simply as a long character string.

Question The current question also stored as a character string;.

The Algorithm :

To answer a question, do the following :

1. Compare each element of QuestionPatterns against the Question and use all those that match successfully to generate a set of text patterns.

2. Pass each of these patterns through a substitution process that generates alternative forms of verbs so that, for example. "go" in a question might match "went" in the text. This step generates a new, expanded set of text patterns.

3. Apply each of these text patterns text patterns to Text, and collect all the resulting answers.

4. Reply with the set of answers just collected.

Examples :

Q.1: The template "What did x y" matches this question and generates the text pattern "Mary go shopping for z." After the pattern-substitution step, this pattern is expanded to a set of patterns including "Mary goes shopping for z," and "Mary went shopping for z." The latter pattern matches the input text; the program, using a convention that variables match the longest possible string up to a sentence delimiter (such as a period), assigns z the value, "new coat," which is given as the answer.

Q.2: Unless the template set is very large, allowing for the insertion of the object of "find" between it and the modifying phrase "that she liked," the insertion of the word "really" in the text, and the substitution of "she" for "Mary," this question is not answerable. If all of these variations are accounted for and the question can be

answered, then the response is "a red one."

Q.3: Since no answer to this question is contained in the text, no answer will be found.

Program 2

This program first converts the input text into a structured internal form that attempts to capture the meaning of the sentences. It also converts questions into that form. It finds answers by matching structured forms against each other.

Data Structures :

English Know A description of the words, grammar, and appropriate semantic interpretations of a large enough subset of English to account for the input texts that the system will see. This knowledge of English is used both to map input sentences into an internal, meaning-oriented form and to map from such internal forms back into English. The former process is used when English text is being read; the latter is used to generate English answers from the meaning-oriented form that constitutes the program's knowledge base.

Input Text The input text in character form.

Structured Text A structured representation of the content of the input text. This structure attempts to capture the essential knowledge contained in the text, independently of the exact way that the knowledge was stated in English. Some things that were not explicit in the English text, such as the referents of pronouns, have been made explicit in this form. Representing knowledge such as this is an important issue in the design of almost all AI programs. Existing programs exploit a variety of frameworks for doing this. There are three important families of such knowledge representation systems; production rules (of the form "if x then y"), slot-and-filler structures, and statements in mathematical logic. Consider an example of a slot-and-filler structure, the sentence "She found a red one she really liked," might be represented as shown in Figure. Actually, that is a simplified description of the contents of the

sentence. Notice that it is not very explicit about temporal relationships (for example, events are just marked as past tense) nor have we made any real attempt to represent the meaning of the qualifier "really." It should, however, illustrate the basic form that representing such as this take. One of the key ideas in this sort

Event 2
- Instance: Finding
- tense: Past
- agent: Mary
- object: Thing1

Thing 1
- instance: Coat
- color: Red

Event 2
- instance: Liking
- tense: Past
- modifier: Much
- object: Thing 1

Figure 1: A Structured Representation of a Sentence

of representation is that entities in the representation derive their meaning from their connections to other entities. In the figure, only the entities defined by the sentence are shown. But other entities corresponding to concepts that the program knew about before it read this sentence, also exist in the representation and can be referred o within these new structures. In this example, for instance, we refer to the entities Mary, Coat (the general concept of a coat of which Thing 1 is a specific instance), Liking (the general concept of liking), and Finding (the general concept of finding).

InputQuestion The input question in character form.

StructQuestion A structured representation of the content of the user's question. The structure is the same as the one used to represent the content of the input text.

The Algorithm :

Convert the InputText into structured form using the knowledge contained in EnglishKnow. This may require considering

several different potential structures, for a variety of reasons, including the fact that English words can be ambiguous, English grammatical structures can be ambiguous, and pronouns may have several possible antecedents.

Then, to answer a question, do the following :

1. Convert the question to structured form, again using the knowledge contained in EnglishKnow. Use some special marker in the structure to indicate the part of the structure that should be returned as the answer. This marker will often correspond to the occurrence of a question word (like "who" or "what") in the sentence. The exact way in which this marking gets done depends on the form chosen for representing StructuredText. If a slot-and-filler structure, such as ours, is used, a special marker can be placed in one or more slots. If a logical system is used, however, markers will appear as variables in the logical formulas that represent the question.

2. Match this structured form against StructuredText.

3. Return as the answer those parts of the text that match the requested segment of the question.

Examples :

Q.1: This question is answered straightforwardly with, "a new coat."

Q.2: This one also is answered successfully with, "a red coat."

Q.3: This one, though, cannot be answered, since there is no direct response to it in the text.

Comments :

This approach is substantially more meaning (knowledge)-based than that of the first program and so is more effective. It can answer most questions to which replies are contained in the text, and it is much less brittle than the first program with respect to the exact forms of the text and the questions. As we expect, based on our experience with the pattern recognition and tic-tac-toe programs, the price we pay for this increased flexibility is time spent searching the various knowledge bases (i.e., EnglishKnow, StructuredText).

Program 3

This program converts the input text into a structured form that contains the meanings of the sentences in the text, and then it combines that form with other structured forms that describe prior knowledge about the objects and situations involved in the text. It answers questions using this augmented knowledge structure.

Data Structures :

WorldModel A structured representation of background world knowledge. This structure contains knowledge about objects, actions, and situations that are described in the input text. This structure is used to construct IntegratedText from the input text. For example, Figure shows an example of a structure that represents the system's knowledge about shopping. This kind of stored knowledge about stereotypical events is called a script. The prime notation describes an object of the same type as the unprimed symbol that may or may not refer to the identical object. In the case of our text, for example, M is a coat and M′ is a red coat. Branches in the figure describe alternative paths through the script.

EnglishKnow Same as in Program 2.

InputText The input text in character form.

IntegratedText A structured representation of the knowledge contained in the input text (similar to the structured description of Program 2) but combined now with other background, related knowledge.

InputQuestion The input question in character form.

StructQuestion A structured representation of the question.

The Algorithm :

Convert the Input Text into structured form using both the knowledge contained in EnglishKnow and that contained in World-Model. The number of possible structures will usually be greater now than it was in Program 2 because so much more knowledge is being used. Sometimes, though, it may be possible to consider fewer

possibilities by using the additional knowledge to filter the alternatives.

Shopping Script :

roles : C (customer), S (salesperson)

props : M (merchandise), D (dollars)

location : L (a store)

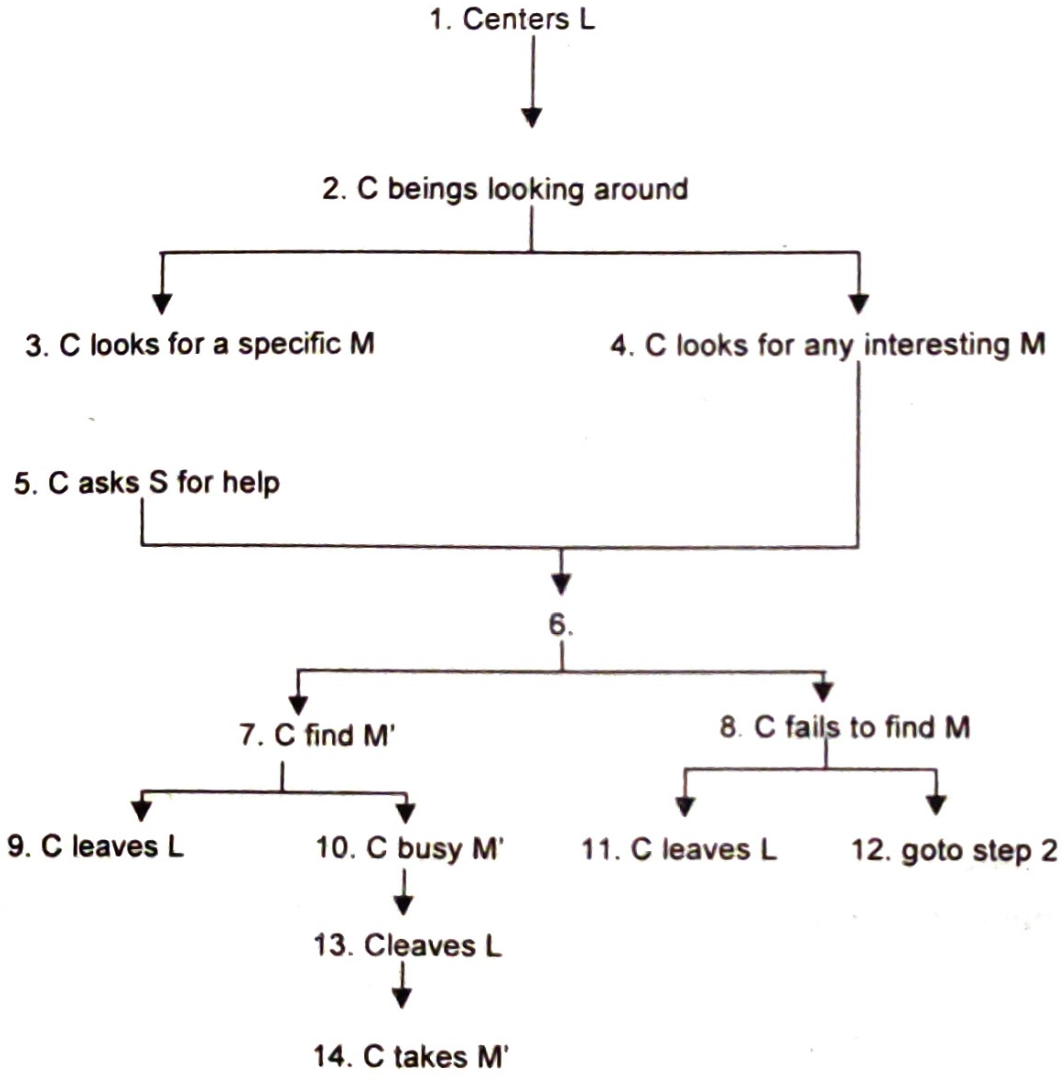

Fig. A shopping Script

To answer a question, do the following :

1. Convert the question to structured form as in Program 2 but use WorldModel if necessary to resolve any ambiguities that may arise.

2. Match this structured form against IntegratedText.

3. Return as the answer those parts of the text that match the requested segment of the question.

*It permits us to define the process of solving a particular problem as a combination of known techniques (each represented as a rule defining a single step in the space) and search, the general technique of exploring the space to try to find some path from the current state to a goal state. Search is a very important process in the solution of hard problems for which no more direct techniques are available.

Q.3. You are given two jugs, a 4 gallon one and 3 gallon one. Neither has any measuring marks on it. There is a pump that can be used to fill jugs with water. Get exactly 2 gallons of water into 4 gallon jug. Initially both jugs are empty.

(a) What are initial and goal states of problem?

(b) What is state space of the problem?

(c) List the set of rules which govern the problem.

(d) Draw an illustrative tree of the solution process.

[W-01]

Ans. The state space for this problem can be described as the set of ordered pairs of integers (x, y), such that x = 0, 1, 2, 3, or 4 and y = 0, 1, 2, or 3; x represents the number of gallons of water in the 3-gallon jug. The start state is (0, 0). The goal state is (2, n) for any value of n (since the problem does not specify how many gallons need to be in the 3-gallon jug).

The operators to be used to solve the problem can be described as shown in Figure 1. Notice that in order to describe the operators completely, it was necessary to make explicit some assumptions not mentioned in the problem statement. We have assumed that we can fill a jug from the pump, that we can pour water out of a jug onto the ground, that we can pour water from one jug to another, and that there are always required when converting from a typical problem statement given in English to a formal representation of the problem suitable for use by a program.

To solve the water jug problem, all we need, in addition to the problem description given above, is a control structure that loops

through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to a goal state. As long as it does not, the cycle continues. Clearly the speed with which the problem gets solved depends on the mechanism that is used to select the next operation to be performed.

For the water jug problem, as with many others, there are several sequences of operators that solve the problem. One such sequence is shown in Figure 2. Often, a problem contains the explicit or implied statement that the shortest (or cheapest) such sequence be found. If present, this requirement will have a significant effect on the choice of an appropriate mechanism to guide the search for a solution.

Several issues that often arise in converting an informal problem statement into a formal problem description are illustrated by this sample water jug problem. The first of these issues concerns the role of the conditions that occur in the left sides of the rules. All but one of the rules shown in Figure1, contain conditions that must be satisfied before the operator described by the rule can be applied. For example, the first rule says, "If the 4-gallon jug is not already full, fill it." This rule could, however, have been written as, "Fill the 4-gallon jug," since it is physically possible to fill the jug even if it is already full. It is stupid to do so since no change in the problem state results, but it is possible. By encoding in the left sides of the rules constraints that are not strictly necessary but that restrict the application of the rules to states in which the rules are most likely to lead to a solution, we can generally increase the efficiency of the problem-solving program that uses the rules.

Each entry in the move vector corresponds to a rule that describes an operation. The left side of each rule describes a board configuration and is represented implicitly by the index position.

The right side of each rule describes the operation to be performed and is represented by a nine-element vector that corresponds to the resulting board configuration. Each of these rules is maximally specific; it applies only to a single board configuration, and as a result, no search is required when such rules are used. However, the drawback to this extreme approach is that the problem solver can take no action at all in a novel situation. In fact, essentially no problem solving ever really occurs. For a tic-tac-toe playing program, this is not a serious problem, since it is possible to enumerate all the situations (i.e., board configurations) that may occur. But for most problems, this is not the case. In order to solve new problems, more general rules must be available.

1	(x, y) if x < 4	→ (4, y)	Fill the 4-gallon jug
2	(x, y) if y < 3	→ (x, 3)	Fill the 3-gallon jug
3	(x, y) if x > 0	→ (x – d, y)	Pour some water out of the 4-gallon jug
4	(x, y) if y > 0	→ (x, y – d)	Pour some water out of the 3-gallon jug.
5	(x, y) if x > 0	→ (0, y)	Empty the 4-gallon jug on the ground
6	(x, y) if y > 0	→ (x, 0)	Empty the 3-gallon jug on the ground
7	(x, y) if x + y ≥ 4 and y > 0	→ (4, y – (4 – x))	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if x + y ≥ 3 and x > 0	→ (x – (3 – y), 3)	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if x + y ≤ 4 and y > 0	→ (x + y, 0)	Pour all the water from the 3-gallon jug into the 4-gallon jug

10	(x, y) if x + y ≤ 3 and x > 0	→ (0, x + y)	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	(0, 2)	→ (2, 0)	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	(2, y)	→ (0, y)	Empty the 2 gallons in the 4-gallon jug on the ground

Fig.1 Production Rules for the Water Jug Problem.

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallong Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5 or 12
0	2	
		9 or 11
2	0	

Fig. One Solution to the Water Jug Problem

A second issue is exemplified by rules 3 and 4 in Figure1. Should they or should they not be included in the list of available operators? Emptying an unmeasured amount of water onto the ground is certainly allowed by the problem statement. But a superficial preliminary analysis of the problem makes it clear that doing so will never get us any closer to a solution. Again, we see the tradeoff between writing a set of rules that describe just the problem itself, as opposed to a set of rules that describe both the problem and some knowledge about its solution.

Rules 11 and 12 illustrate a third issue. To see the problem-solving knowledge that these rules represent, look at the last two steps of the solution shown in Figure2. Once the state (4, 2) is reached, it is obvious what to do next. The desired 2 gallons have been produced, but they are in the wrong jug. So the thing to do is to move them (rule 11). But before that can be done, the water that is already in the 4-gallon jug must be emptied out (rule 12). The idea behind these special-purpose rules is to capture the special-case knowledge that can be used at this stage in solving the problem. These rules do not actually add power to the system since the operations they describe are already provided by rule 9 (in the case of rule 11) and by rule 5 (in the case of rule 12). In fact, depending on the control strategy that is used for selecting rules to use during problem solving, the use of these rules may degrade performance. But the use of these rules may also improve performance if preference is given to special-case rules.

Q.4. Write short note on production system. *[W-01]*

Ans. Since search forms the core of many intelligent processes, it is useful to structure AI programs in a way that facilitates describing and performing the search process. Production systems provide such structures. A definition of a production system is given below. Do not be confused by other uses of the word production, such as to describe what is done in factories. A production system consists of

* A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.

* One or more knowledge/databases that contain whatever information is appropriate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.

* A control strategy that specifies the order in which the rules

will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.

*A rule applier.

It encompasses a great many systems, including our descriptions of both a chess player and a water jug problem solver. It also encompasses a family of general production system interpreters, including :

* Basic production system languages, such as OPS5 and ACT.

* More complex, often hybrid systems called expert system shells, which provide complete (relatively speaking) environments for the construction of knowledge-based expert systems.

* General problem-solving architectures like SOAR, a system based on a specific set of cognitively motivated hypotheses about the nature of problem solving.

All of these systems provide the overall architecture of a production system and allow the programmer to write rules that define particular problems to be solved.

We have now seen that in order to solve a problem, we must first reduce it to one for which a precise statement can be given. This can be done by defining the problem's state space (including the start and goal states) and a set of operators for moving in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modeled as a production system.

Q.5. Explain in detail control strategies.

OR Write short note on Back tracking. *[S-97, W-97]*

OR Differentiate between Breadth first and Depth first search. *[S-98]*

Ans. The first requirement of a good control strategy is that it cause motion. Consider again the water jug problem of the last section. Suppose we implemented the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, we would never solve the problem.

We would continue indefinitely filling the 4-gallon jug with water. Control strategies that do not cause motion will never lead to a solution.

The second requirement of a good control strategy is that it be systematic. Here is another simple control strategy for the water jug problem : On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. But we are likely to arrive at the same state several times during the process and to use many more steps than are necessary. Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution. The requirement that a control strategy be systematic corresponds to the need for global motion (over the course of several steps) as well as for local motion (over the course of a single step). One systematic control strategy for the water jug problem is the following. Construct a tree with the initial state as its root. Generate all the offspring of the root by applying each of the applicable rules to the initial state. Figure 1 shows how the tree looks at this point. Now for each leaf node, generate all its successors by applying all the rules that are appropriate. The tree at this point is shown in Figure. Continue this process until some rule produces a goal state. This process, called breadth-first search, can be described precisely as follows.

Algorithm : Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state.

2. Until a goal state is found or NODE-LIST is empty do :

(a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit.

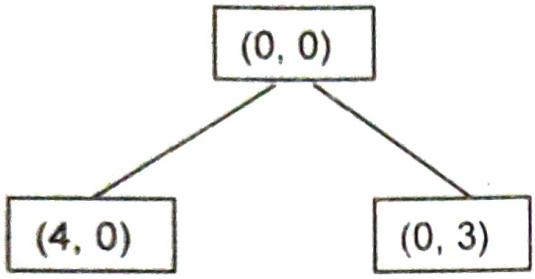

Fig. One Level of a Breadth-First Search Tree

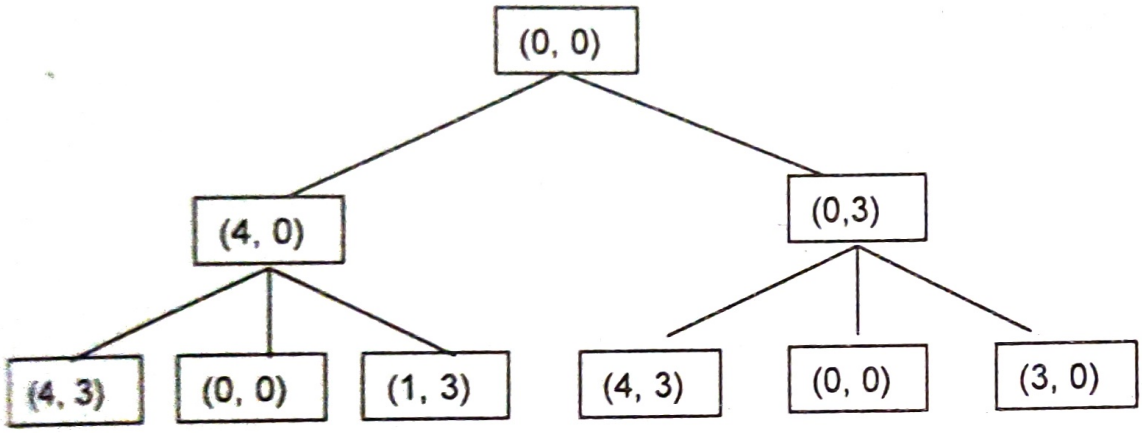

Fig. Two Levels of a Breadth-First Search Tree

(b) For each way that each rule can match the state described in E, do :

i. Apply the rule to generate a new state.

ii. If the new state is a goal state, quit and return this state.

iii. Otherwise, add the new state to the end of NODE-LIST.

Other systematic control strategies are also available. For example, we could pursue a single branch of the tree until it yields a solution or until a decision to terminate the path is made. It makes sense to terminate a path if it reaches a dead-end, produces a previous state, or becomes longer than some prespecified "futility" limit. In such a case, backtracking occurs. The most recently created state from which alternative moves are available will be revisited and a new state will be created. This form of backtracking is called chronological backtracking because the order in which steps are undone depends only on the temporal sequence in which the steps were originally made. Specifically, the most recent step is always the first to be undone. This form of backtracking is what is usually meant by the simple term backtracking. But there are other

problem. If it is very likely that the same node will be generated in several different ways, then it is more worthwhile to use a graph procedure than if such duplication will happen only rarely.

Graph search procedures are especially useful for dealing with partially commutative production systems in which a given set of operations will produce the same result regardless of the order in which the operations are applied. A systematic search procedure will try many of the permutations of these operators and so will generate the same node many times. This is exactly what happened in the water jug example shown above.

Q.9. List different heuristic search techniques & explain any one.

(1) Depth-first search.

(2) Breadth-first search

(3) Generate-and-Test.

(4) Hill climbing.

(5) Best-first search.

(6) Problem reduction.

(7) Constraint specification.

(8) Means-ends analysis. ***[S-98]***

Ans. Algorithm : Generate-and-Test

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.

2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.

3. If a solution has been found, quit. Otherwise, return to step 1.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately, if the problem space is very large, "eventually" maybe a very long time.

The generate-and-test algorithm is a depth-first search procedure since complete solutions must be generated before they can be tested. In its most systematic form, it is simply an exhaustive search of the problem space. Generate-and-test can also operate by generating solutions randomly, but then there is no guarantee that a solution will ever be found.

The most straightforward way to implement systematic generate-and-test is as a depth-first search tree with backtracking. If some intermediate states are likely to appear often in the tree, however, it may be better to modify that procedure, as described above, to traverse a graph rather than a tree.

For simple problems, exhaustive generate-and-test is often a reasonable technique. For example, consider the puzzle that consists of four six-sided cubes, with each side of each cube painted on of four colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row one block face of each color is showing. This problem can be solved by a person (who is a much slower processor for this sort of thing than even a very cheap computer) in several minutes by systematically and exhaustively trying all possibilities. It can be solved even more quickly using a heuristic generate-and-test procedure. A quick glance at the four blocks reveals that there are more, say, red faces than there are of other colors. Thus when placing a block with several red faces, it would be a good idea to use as few of them as possible as outside faces. As many of them as possible should be placed to abut the next block. Using this heuristic, many configurations need never be explored and a solution can be found quite quickly.

Q.10. Discuss the steepest ascent hill climbing search technique and how to overcome the hurdles like local maxima, foothill and plateau. *[W-99]*

OR Discuss some of the potential problems using hill climbing search. Give examples of the problem cited *[S-01, W-03]*

OR State the difference between simple hill climbing and steepest ascent hill climbing. *[W-01]*

Ans. 1. Simple Hill Climbing

Algorithm : Simple Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Loop until a solution is found or until there are no new operators left to be applied in the current state :

(a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(b) Evaluate the new state.

i. If it is goal state, then return it and quit.

ii. If it is not a goal state but it is better than the current state, then make it the current state.

iii. If it is not better than the current state, then continue in the loop.

The key difference between this algorithm and the one we gave for generate-and-test is the use of an evaluation function as a way to inject task-specific knowledge into the control process.

Notice that in this algorithm, we have asked the relatively vague question, "Is one state better than another?" For the algorithm to work, a precise definition of better must be provided. In some cases, it means a higher value of the heuristic function. In others, it means a lower value. It does not matter which, as long as a particular hill-climbing program is consistent in its interpretation.

To see how hill climbing works, consider puzzle of the four colored blocks. To solve the problem, we first need to define a heuristic function that describes how close a particular configuration is to being a solution. One such function is simply the sum of the number of different colors on each of the four sides. A solution to

the puzzle will have a value of 16. Next we need to define a set of rules that describe ways of transforming one configuration into another. Actually, one rule will suffice. It says simply pick a block and rotate it 90 degrees in any direction. Having provided these definitions, the next step is to generate a starting configuration. This can either be done at random or with the aid of the heuristic function. Now hill climbing can begin. We generate a new state by selecting a block and rotating it. If the resulting state is better, then we keep it. If not, we return to the previous state and try a different perturbation.

2. Steepest-Ascent Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called steepest-ascent hill climbing or gradient search. Notice that this contrasts with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows.

Algorithm : Steepest-Ascent Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Loop until a solution is found or until a complete iteration produces no change to current state :

(a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC.

(b) For each operator that applies to the current state do :

i. Apply the operator and generate a new state.

ii. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.

(c) If the SUCC is better than current state, then set current state to SUCC.

To apply steepest-ascent hill climbing to the colored blocks problem, we must consider all perturbations of the initial state and

choose the best. For this problem, this is difficult since there are so many possible moves. There is a trade-off between the time required to select a move (usually longer for steepest-ascent hill climbing) and the number of moves required to get to a solution (usually longer for basic hill climbing) that must be considered when deciding which method will work better for a particular problem.

Both basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau, or a ridge.

A local maximum is a state that is better than all its neighbors but is not better than some other states farther away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called foothills.

A plateau is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

A ridge is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.

There are some ways of dealing with these problems, although these methods are by no means guaranteed :

*Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path

that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.

* Make a big jump in some direction to try to get to new section of the search space. This is a particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.

* Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a particularly good strategy for dealing with ridges.

Hill climbing is a local method, by which we mean that it decides what to do next by looking only at the "immediate" consequences of its choice rather than by exhaustively exploring all the consequences. It shares with other local methods, such as the nearest neighbor heuristic the advantage of being less combinatorially explosive than comparable global methods. But it also shares with other local methods a lack of a guarantee that it will be effective. Although it is true that the hill-climbing procedure itself looks only one move ahead and not any farther, that examination may in fact exploit an arbitrary amount of global information if that information is encoded in the heuristic function. Consider the blocks world problem shown in Figure 1. Assume the same operators (i.e., pick up one block and put it on the table; pick up one block and put it on another one). Suppose we use the following heuristic function :

Local : Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

Using this function, the goal state has a score of 8. The initial state has a score of 4 (since it gets one point added for blocks C, D, E, F, G, and H and one point subtracted for blocks A and B). There is only one move from the initial state, namely to move block A to the table. That produces a state with a score of 6 (since now A's position causes a point to be added rather than subtracted). The hill-

climbing procedure will accept that move. From the new state, there are three possible moves, leading to the three states shown in Figure 1. These states have the scores : (a) 4, (b) 4, and (c) 4. Hill climbing will halt because all these states have lower scores than the current state. The process has reached a local maximum that is not the global maximum. The problem is that by purely local examination of support structures, the current state appears to be better than any of its successors because more blocks rest on the correct objects. To solve this problem, it is necessary to disassemble a good local structure (the stack B through H) because it is in the wrong global context.

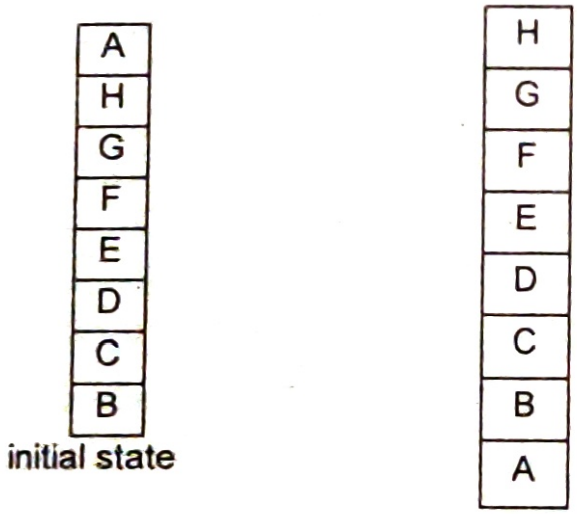

Fig. A Hill-Climbing Problem

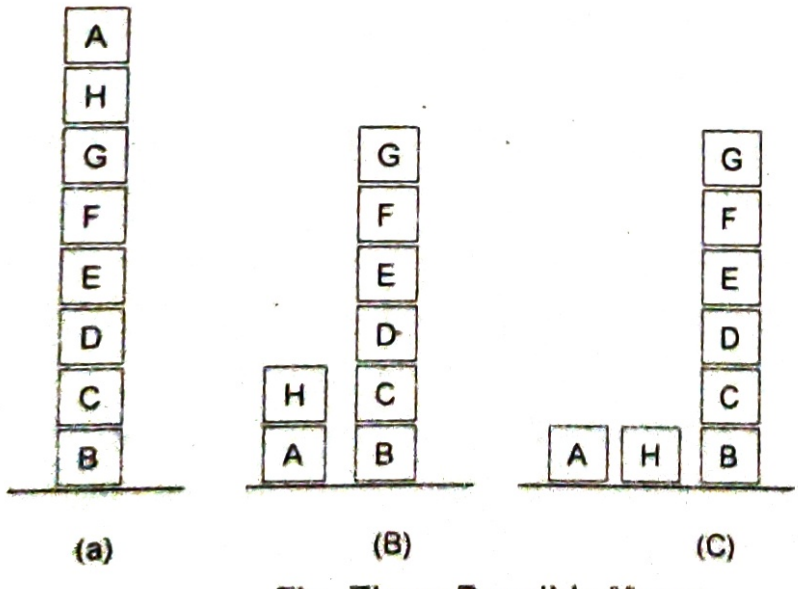

Fig. Three Possible Moves

We could blame hill climbing itself for this failure to look far enough ahead to find a solution. But we could also blame the heuristic function and try to modify it. Suppose we try the following heuristic function in place of the first one :

Global : For each block that has the correct support structure (i.e., the complete structure underneath it is exactly as it should be), add one point for every block in the support structure. For each block that has an incorrect support structure, subtract one point for every block in the existing support structure.

Using this function, the goal state has the score 28 (1 for B, 2 for C, etc). The initial state has the score – 28. Moving A to the table yields a state with a score of – 21 since A no longer has seven wrong blocks under it. The three states that can be produced next now have the following scores : (a) – 28, (b) – 16, and (c) – 15. This time, steepest-ascent hill climbing will choose move (c), which is the correct one. This new heuristic function captures the two key aspects of this problem : incorrect structures are bad and should be taken apart : and correct structures are good and should be built-up. As a result, the same hill climbing procedure that failed with the earlier heuristic function now works perfectly.

3. Simulated Annealing

Simulated annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made. The idea is to do enough exploration of the whole space early on so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau, or a ridge.

The algorithm for simulated annealing is only slightly differences from the simple hill-climbing. The three differences are :

* The annealing schedule must be maintained.

* Moves to worse states may be accepted.

* It is a good idea to maintain, in addition to the current state, the best state found so far. Then, if the final state is worse than that earlier state (because of the luck in accepting moves to worse states), the earlier state is still available.

Algorithm : Simulated Annealing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Initialize BEST-SO-FAR to the current state.

3. Initialize T according to the annealing schedule.

4. Loop until a solution is found or until there are no new operators left to be applied in the current state.

(a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(b) Evaluate the new state. Compute

ΔE = (value of current) – (value of new state)

* If the new state is a goal state, then return it and quit.

* If it is not a goal state but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.

* If it is not better than the current state, then make it the current state with probability p′ as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0, 1]. If that number is less than p′ then the move is accepted. Otherwise, do nothing.

(c) Revise T as necessary according to the annealing schedule.

5. Return BEST-SO-FAR as the answer.

Q.11. Discuss Best-first search method.

OR

When would best-first search be worse than simple breadth-first search? *[W-98, W-99]*

OR

Discuss and compare hill climbing and best first search technique. *[W-00, W-02]*

OR

Explain A* algorithm in detail. *[W-01, W-03]*

Ans. Best-first search, is a way of combining the advantages of both depth-first and breadth-first search into a single method.

the exact sum of the costs of applying each of the rules that were applied along the best path to the node. The function h' is an estimate of the additional cost of getting from the current node to a goal state. This is the place where knowledge about the problem domain is exploited. The combined function f', them, represents an estimate of the cost of getting from the initial state to a goal state along the path that generate the current node. If more than one path generated the node, then the algorithm will record the best one. Note that because g and h' must be added, it is important that h' be a measure of the cost of getting from the node to a solution (i.e., good nodes get low values; bad nodes get high values) rather than a measure of the goodness of a node (i.e., good nodes get high values).

The actual operation of the algorithm is very simple. It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor. Then the next step begins.

This process can be summarized as follows :

Algorithm : Best-First Search

1. Start with OPEN containing just the initial state.

2. Until a goal is found or there are no nodes left on OPEN do :

(a) Pick the best node on OPEN.

(b) Generate its successors.

(c) For each successor do :

i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.

ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that node may already have.

Algorithm : A*

1. Start with OPEN containing only the initial node. Set that node's value to 0, its h' value to whatever it is, and its f' value to h' + 0, or h'. Set CLOSED to the empty list.

2. Until a goal node is found, repeat the following procedure : If there are no nodes on OPEN, report failure. Otherwise, pick the node on OPEN with the lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED. See if BESTNODE is a goal node. If so, exit and report a solution (either BESTNODE if all we want is the node or the path that has been created between the initial state and BESTNODE if we are interested in the path). Otherwise, generate the successors of BESTNODE but do not set BESTNODE to point to them yet. (First we need to see if any of them have already been generated.) For each such SUCCESSOR, do the following :

(a) Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.

(b) Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR.

(c) See if SUCCESSOR is the same as any node on OPEN (i.e., it has already been generated but not processed). If so, call that OLD. Since this node already exists in the graph, we can throw SUCCESSOR away and add OLD to the list of BESTNODE's successors. Now we must decide whether OLD's parent link should be reset to point to BESTNODE. It should be if the path we have just found to SUCCESSOR is cheaper than the current best path to OLD (since SUCCESSOR and OLD are really the same node). So see whether it is cheaper to get to OLD via its current parent or to SUCCESSOR via BESTNODE by comparing their g values. If OLD is cheaper (or just as cheap), then we need do nothing. If SUCCESSOR is cheaper, then reset OLD's parent link to point to BESTNODE, record the new cheaper path in g(OLD),and update f'(OLD).

(d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors. Check to see if the new path or the old path is better just as in step 2(c), and set the parent link and g and f′ values appropriately. If we have just found a better path to OLD, we must propagate the improvement to OLD's successors. This is a bit tricky. OLD points to its successors. Each successor in turn point to its successors, and so forth, until each branch terminates with a node that either is still on OPEN or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at OLD, changing each node's g value (and thus also its f′ value), terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found This condition is easy to check for. Each node,s parent link points back to its best known parent. As we propagate own to a node, see if its parent points to the node we are coming from. If so, continue the propagation. If not, then its g value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of g being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation.

(e) If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. Compute f′ (SUCCESSOR) = g(SUCCESSOR) + h′ (SUCCESSOR).

Several interesting observation can be made about this algorithm. The first concerns the role of the g function. It lets us choose which node to expand next on the basis not only of how good the node itself looks (as measured by h′), but also on the basis of

how good the path to the node was. By incorporating g into f′, we will not always choose as our next node to expand the node that appears to be closest to the goal. This is useful if we care about the path we find. If, on the other hand, we only care about getting to a solution somehow, we can define g always to be 0, thus always choosing the node that seems closest to a goal. If we want to find a path involving the fewest number of steps, then we set the cost of going from a node to its successor as a constant, usually 1. If, on the other hand, we want to find the cheapest path and some operators cost more than others, then we set the cost of going from one node to another to reflect those costs. Thus the A* algorithm can be used whether we are interested in finding a minimal-cost overall path or simply and path as quickly as possible.

The second observation involves h′, the estimator of h′, the distance of a node to the goal. If h′ is a perfect estimator of h, then A* will converge immediately to the goal with no search. The better h′ is, the closer we will get to that direct approach. If, on the other hand, the value of h′ is always 0, the search will be controlled by g. If the value of g is also 0, the search strategy will be random. If the value of g is always 1 , the search will be breadth first. All nodes on the next level. We can guarantee that h′ never overestimates h. In that case, the A* algorithm is guaranteed to find an optimal (as determined by g) path to a goal, if one exists. This can easily be seen from a few examples.

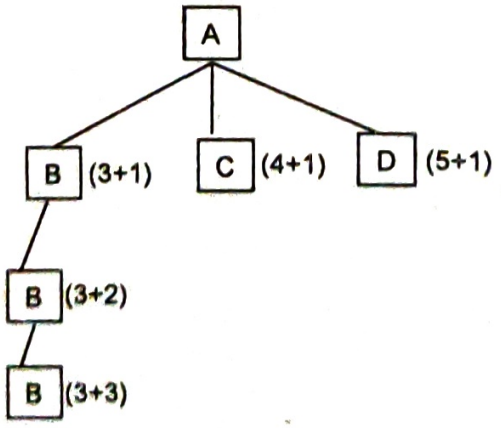

Fig. h′ Underestimates h

Consider the situation shown in Figure 2. Assume that the cost of all arcs is 1. Initially, all nodes except A are on OPEN (although the figure shows the situation two steps later, after B and E have been expanded). For each node, f′ is indicated as the sum of h′ and g. In this example, node B has the lowest f′, 4, so it is expanded first. Suppose it has only one successor E, which also appears to be three moves away from a goal. Now f′(E) is 5, the same as f′(C). Suppose we resolve this in favor of the path we are currently following. Then we will expand E next. Suppose it too has a single successor F, also judged to be three moves from a goal. Few are clearly using up moves and making no progress. But f′(F) = 6, which is greater than f′(C). So we will expand C next. Thus we see that by underestimating h(B) we have wasted some effort. But eventually we discover that B was farther away than we thought and we go back and try another path.

Now consider the situation shown in Figure . Again we expand B on the first step. On the second step we again expand E. At the next step we expand F, and finally we generate G, for a solution path of length 4. But suppose there is a direct path from D to a solution, giving a path of length 2. We will never find it. By overestimating h′(D) we make D look so bad that we may find some other, worse solution without ever expanding D. In general, if h′ might overestimate h, we cannot be guaranteed of finding the cheapest path solution unless we expand the entire graph until all paths are longer than the best solution.

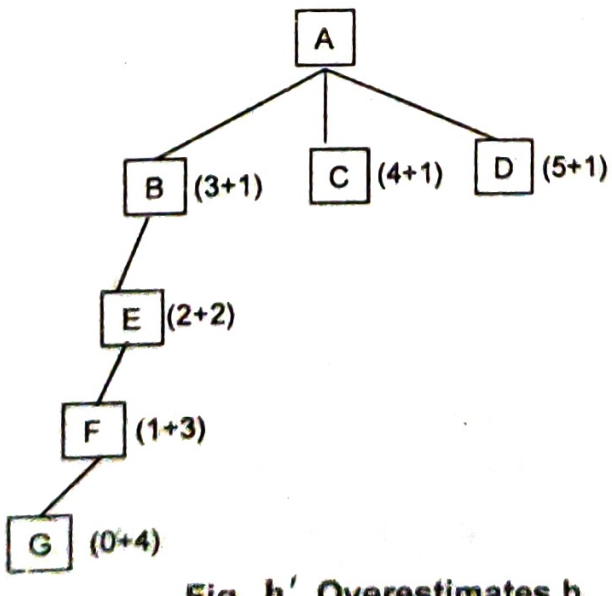

Fig. h′ Overestimates h