INTELLIGENT AGENTS

VI SEMESTER

UNIT- I

UNIT - 1

Problems and search: What is AI: AI problems, underlying assumption, AI technique, level of model, Problems, Problem spaces and search: Defining the problem as a state space search, production systems, problem characteristics, production system characteristics, Issues in the design of search programs, Heuristic search techniques, generate-and-test, hill climbing

1.1 What is AI

1.1.1 AI Problems

1.1.2 Underlying assumption

1.1.3 AI technique

1.1.4 Level of model

1.2 Problems, Problem spaces and search

1.2.1 Defining the problem as a state space search

1.2.2 Production systems

1.2.3 Problem characteristics

1.2.4 Production System characteristics

1.2.5 Issues in the design of search programs

1.3 Heuristic Search Techniques

1.3.1 Generate and Test

1.3.2 Hill Climbing

1.1 WHAT IS AI:

Artificial Intelligence(AI) is the study of how to make computers do things which, at the moment, people do better.

1.1.1 AI PROBLEMS:

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. 1.1 *Some of the Task Domains of Artificial Intelligence*

The above Figure 1.1 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, Mundane(General) tasks of perceptual, linguistic and common sense skills are learned. Later, 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(controlled) 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 than do the more mundane skills and that knowledge is usually easier to represent and deal with inside programs.

As a result, the problem areas where AI is now flourishing most as a practical discipline are primarily the domains that require only specialized expertise without the assistance of common sense 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.

1.1.2 UNDERLYING ASSUMPTION:

At the heart of research in artificial intelligence lies what Newell and Simon [1976] call the *physical symbol system hypothesis.* They define a physical symbol system as follows:

> 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.

They then state the hypothesis as

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

This hypothesis is only a hypothesis. There appears to be no way to prove or disprove it on logical grounds. So it must be subjected to empirical validation. We may find that it is false. We may find that the bulk of the evidence says that it is true. But the only way to determine its truth is by experimentation.

1.1.3 WHAT IS AN AI TECHNIQUE:

AI research says that “Intelligence requires knowledge”. Knowledge possesses some properties as follows:

i) It is voluminous.

ii) It is hard to characterize accurately.

iii) It is constantly changing.

iv) It differs from data by being organized in a way that corresponds to the ways it will be used.

AI technique is a method that exploits knowledge that should be represented in such a way that:

i) 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.

ii) It can be understood by people who must provide it.

iii) It can easily be modified to correct errors and to reflect changes in the world and in our world view.

iv) It can be used in a great many situations even if it is not totally accurate or complete.

v) It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must usually be considered.

EXAMPLES:

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 blank), 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 an 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	If 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.

2) Question Answering (NLP)

In this section, we look at 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.

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:

Q1: What did Mary go shopping for?
Q2: What did Mary find that she liked?
Q3: 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 text.

Data Structures

QuestionPatterns 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 to Text, and collect all the resulting answers.
4. Reply with the set of answers just collected.

Examples

Q1: 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, "a new coat," which is given as the answer.

Q2: 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."

Q3: Since no answer to this question is contained in the text, no answer will be found.

Comments

This approach is clearly inadequate to answer the kinds of questions people could answer after reading a simple text. Even its ability to answer the most direct questions is delicately dependent on the exact form in which questions are stated and on the variations that were anticipated in the design of the templates and the pattern substitutions that the system uses. In fact, the sheer inadequacy of this program to perform the task may make you wonder how such an approach could even be proposed. This program is substantially farther away from being useful than was the initial program we looked at for tic-tac-toe. Is this just a strawman designed to make some other technique look good in comparison? In a way, yes, but it is worth mentioning that the approach that this program uses, namely matching patterns, performing simple text substitutions, and then forming answers using straightforward combinations of canned text and sentence fragments located by the matcher, is the same approach that is used in one of the most famous "AI" programs ever written—ELIZA, which we discuss in Section 6.4.3. But, as you read the rest of this sequence of programs, it should become clear that what we mean by the term "artificial intelligence" does not include programs such as this except by a substantial stretching of definitions.

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

EnglishKnow — 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.

InputText — The input text in character form.

StructuredText — 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. We discuss all of these methods later in substantial detail, and we look at key questions that need to be answered in order to choose a method for a particular program'. For now though, we just pick one arbitrarily. The one we've chosen is a slot-and-filler structure. For example, the sentence "She found a red one she really liked," might be represented as shown in Fig. 1.2. Actually, this 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 representations such as this take. One of the key ideas in this sort 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 re- ferred to within these new structures. In this example, for instance, we refer to the entities *Mary*, *Coat* (the general concept of a coat of which *Thing1* is a specific instance), *Liking* (the general concept of liking), and *Finding* (the general concept of finding).

Event 2
- *instance :* *Finding*
- *tense:* *Past*
- *agent :* *Mary*
- *object:* *Thing1*

Thing1
- *instance:* *Coat*
- *color:* *Red*

Event2
- *instance:* *Liking*
- *tense :* *Past*
- *modifier:* *Much*
- *object:* *Thing1*

Fig. 1.2 *A Structured Representation of a Sentence*

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

Q1: This question is answered straightforwardly with, "a new coat".
Q2: This one also is answered successfully with, "a red coat".
Q3: 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).

One word of warning is appropriate here. The problem of producing a knowledge base for English that is powerful enough to handle a wide range of English inputs is very difficult. It is discussed at greater length in Chapter 15. In addition, it is now recognized that knowledge of English alone is not adequate in general to enable a program to build the kind of structured representation shown here. Additional knowledge about the world with which the text deals is often required to support lexical and syntactic disambiguation and the correct assignment of antecedents to pronouns, among other things. For example, in the text

> Mary walked up to the salesperson. She asked where the toy department was.

it is not possible to determine what the word "she" refers to without knowledge about the roles of customers and sales people in stores. To see this, contrast the correct antecedent of 'she" in that text with the correct antecedent for the first occurrence of "she" in the following example:

> Mary walked up to the sales person. She asked her if she needed any help.

In the simple case illustrated in our coat-buying example, it is possible to derive correct answers to our first two questions without any additional knowledge about stores or coats, and the fact that some such additional information may be necessary to support question answering has already been illustrated by the failure of this

program to find an answer to question 3. Thus we see that although extracting a structured representation of the meaning of the input text is an improvement over the meaning-free approach of Program 1, it is by no means sufficient in general. So we need to look at an even more sophisticated (i.e., knowledge-rich) approach, which is what we do next.

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 1.3 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* and is discussed in more detail in Section 10.2. The notation used here differs from the one normally used in the literature for the sake of simplicity. 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.

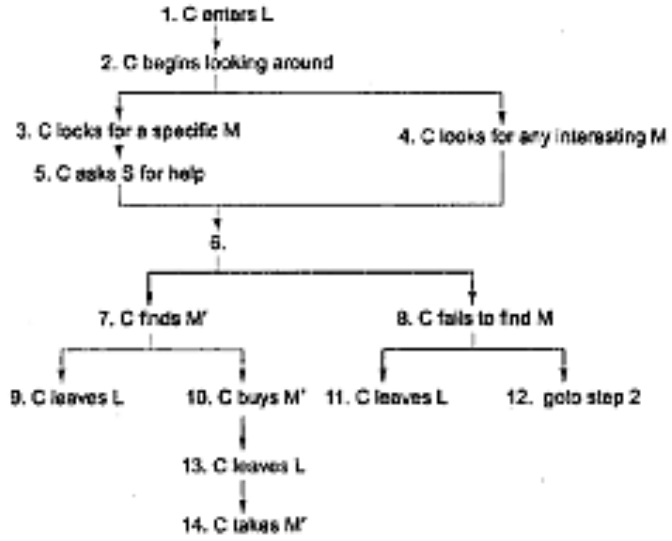


Fig. 1.3 *A Shopping Script*

EnglishKnow — Same as in Program 2.
InputText — The input text in character form.

Why couldn't Mary's brother reach her?

with the reply

Because she wasn't home.

But to do so requires knowing that one cannot be at two places at once and then using that fact to conclude that Mary could not have been home because she was shopping instead. Thus, although we avoided the inference problem temporarily by building IntegratedText, which had some obvious inferences built into it, we cannot avoid it forever. It is simply not practical to anticipate all legitimate inferences. In later chapters, we look at ways of providing a general inference mechanism that could be used to support a program such as the last one in this series.

This limitation does not contradict the main point of this example though. In fact, it is additional evidence for that point, namely, an effective question-answering procedure must be one based soundly on knowledge and the computational use of that knowledge. The purpose of AI techniques is to support this effective use of knowledge.

With the advent of the Internet and the vast amount of knowledge in the ever increasing websites and associated pages, came the Web based Question Answering Systems. Try for instance the START natural language question answering system (http://start.csail.mit.edu/). You will find that both the questions – *What is the capital of India?* and *Is Delhi the capital of India?* yield the same answers, viz. *New Delhi is the capital of India.* On the contrary the question — *Are there wolves in Korea?* yields *I don't know if there are wolves in Korea.* which looks quite natural.

1.3.3 Conclusion

We have just examined two series of programs to solve two very different problems. In each series, the final program exemplifies what we mean by an AI technique. These two programs are slower to execute than the earlier ones in their respective series, but they illustrate three important AI techniques:

- Search—Provides a way of solving problems for which no more direct approach is available as well as a framework into which any direct techniques that are available can be embedded..
- Use of Knowledge—Provides a way of solving complex problems by exploiting the structures of the objects that are involved.
- Abstraction—Provides a way of separating important features and variations from the many unimportant ones that would otherwise overwhelm any process.

1.1.4 LEVEL OF MODEL:

Before we set out to do something, it is a good idea to decide exactly what we are trying to do. So, we must ask ourselves, "What is our goal, in trying to produce programs that do the intelligent things that people do?" Are we trying to produce programs that do the tasks the same way people do? Or, are we attempting to produce programs that simply do the tasks in whatever way appear easiest?

Efforts to build programs that perform tasks the way people do can be divided into two classes:

i) Programs in the first class attempt to solve problems that do not really fit our definition of an AI task.

Example: A classical example of this class of program is the Elementary Perceiver And Memorizer (EPAM) [Feigenbaum, 1963], which memorized associated pairs of nonsense syllabus.

ii) The second class of programs that attempt to model human performance are those that do things that fall more clearly within our definition of AI tasks. There are several reasons, one might want to model human performance at these sorts of tasks:

1) To test psychological theories of human performance. One example of a program that was written for this reason is PARRY [Colby, 1975], which exploited a model of human paranoid behavior to simulate the conversational behavior of a paranoid person. The model was good enough that when several psychologists were given the opportunity to converse with the program via a terminal, they diagnosed its behavior as paranoid.

2) To enable computers to understand human reasoning. For example, for a computer to be able to read a newspaper story and then answer a question, such as "Why did the terrorists kill the hostages?" Its program must be able to simulate the reasoning processes of people.

3) To enable people to understand computer reasoning. In many circumstances, people are reluctant to rely on the output of a computer unless they can understand as to how the machine arrived at its result. If the computer's reasoning process is similar to that of people, then, producing an acceptable explanation is much easier.

4) To exploit what knowledge we can glean (gather) from people. Since, people are the best-known performers of most of the tasks with which we are dealing, it makes a lot of sense to look to them for clues as to how to proceed.

Human cognitive theories have also influenced AI to look for higher-level (i.e., far above the neuron level) theories that do not require massive parallelism for their implementation.

1.2 Problems, Problem spaces and search

1.2.1 Defining the problem as a state space search

State space search

"It is complete set of states including start and goal states, where the answer of the problem is to be searched".

Problem

"It is the question which is to be solved. For solving the problem it needs to be precisely defined. The definition means, defining the start state, goal state, other valid states and transitions".

A state space representation allows for the formal definition of a problem which makes the movement from initial state to the goal state quite easily. So we can say that various problems like planning, learning, theorem proving etc. are all essentially search problems only.

State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or states of an instance are considered, with the goal of finding a *goal state* with a desired property.

Example 1: The eight tile puzzle problem formulation

Example 1: The eight tile puzzle problem formulation

The eight tile puzzle consist of a 3 by 3 (3*3) square frame board which holds 8 movable tiles numbered 1 to 8. One square is empty, allowing the adjacent tiles to be shifted. The objective of the puzzle is to find a sequence of tile movements that leads from a starting configuration to a goal configuration.

1	2	3
4	8	-
7	6	5

Start State

1	2	3
4	5	6
7	8	-

Goal State

1	2	3
4	8	-
7	6	5

Steps:

Initially:

{(1,2,3), (4,8,0), (7,6,5)}

Step 1: {(1,2,3), (4,8,0), (7,6,5)}

1	2	3
4	8	-
7	6	5

Step 2: {(1,2,3), (4,8,5), (7,6,0)}

1	2	3
4	8	5
7	6	-

Step 3: {(1,2,3), (4,8,5), (7,0,6)}

1	2	3
4	8	5
7	-	6

Step 4: {(1,2,3), (4,0,5), (7,8,6)}

1	2	3
4	0	5
7	8	6

Step 5: {(1,2,3), (4,5,0), (7,8,6)}

1	2	3
4	5	6
7	8	0

Step 6: {(1,2,3), (4,5,6), (7,8,0)}

The states of 8 tile puzzle are the different permutations of the tiles within frame.

Lets do a standard formulation of this problem now.

States: It specifies the location of each of the 8 tiles and the blank in one of the nice squares.

Initial state : Any state can be designated as the initial state.

Goal : Many goal configurations are possible one such is shown in the figure

Legal moves (or state) : The generate legal states that result from trying the four actions-

- Blank moves left
- Blank moves right
- Blank moves up
- Blank moves down

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

The eight tile puzzle consist of a 3 by 3 (3*3) square frame board which holds 8 movable tiles numbered 1 to 8. One square is empty, allowing the adjacent tiles to be shifted. The objective of the puzzle is to find a sequence of tile movements that leads from a starting configuration to a goal configuration.

The states of 8 tile puzzle are the different permutations of the tiles within frame.

Lets do a standard formulation of this problem now.

States: It specifies the location of each of the 8 tiles and the blank in one of the nice squares.

Initial state : Any state can be designated as the initial state.

Goal : Many goal configurations are possible one such is shown in the figure

Legal moves (or state) : They generate legal states that result from trying the four actions-

- Blank moves left
- Blank moves right
- Blank moves up
- Blank moves down

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

Example 2: You are given two jugs , a 4- gallon one and a 3- gallon one .Neither has any measuring markers on it . There is a pump that can be used to fill the jugs with water how can you get exact 2 gallons of water into the 4- gallon jug?

The state space for this problem can be described as of ordered pairs of integers (x, y) , such that x=0,1,2,3 or 4 and y=0,1,2 or 3 , represents the number of gallons of water in the 4- gallon jug , and y represents the quantity 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 Fig bellow. They are represented as rules whose left side are matched against the current state and whose right sides describe the new state that results from applying the rule.

1- (x,y) $\longrightarrow$ (4,y) fill the 4- gallon jug
If x<4

2- (x,y) ⟶ (x,3) fill the 3-gallon jug
If y<3

3- (x,y) ⟶ (x-d,y) pour some water out of the 4- gallon jug
If x>0

4- (x,y) ⟶ (x-d,y) pour some water out of the 3- gallon jug
If y>0

5-(x,y) ⟶ (0,y) empty the 4- gallon jug on the ground
If x>0

6-(x,y) ⟶(x,0) empty the 3- gallon jug on the ground
If y>0

7- (x,y) ⟶(4,y-(4-x)) **pour water from the 3- gallon jug into the 4-gallon jug until the 4-galoon jug is full**
If x+y>=4 and y>0

8- (x,y) ⟶(x-(3-y),3)) **pour water from the 4- gallon jug into the 3-gallon jug until the 3-gallon jug is full**
If x+y>=3 and x>0

9- (x,y) ⟶ (x+y,0) **pour all the water from the 3 -gallon jug into the 4-gallon jug**
If x+y<=4 and y>0

10- (x,y) ⟶(0,x+y) **pour all the water from the 4 -gallon jug into the 3-gallon jug**
If x+y<=3 and x>0

11- (0,2) ⟶(2,0) **pour the 2-gallon from the 3 -gallon jug into the 4-gallon jug**

12- (2,y) ⟶(0,x) **empty the 2 gallon in the 4 gallon on the ground**

Production for the water jug problem

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

This is one solution to the water Jug problem.

Breadth First Search (BFS) searches breadth-wise in the problem space. Breadth-First search is like traversing a tree where each node is a state which may a be a potential

candidate for solution. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

Algorithm: Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
    a. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
    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 the goal state, quit and return this state.
iii) Otherwise add this state to the end of NODE-LIST

Advantages of Breadth-First Search

1. Breadth first search will never get trapped exploring the useless path forever.
2. If there is a solution, BFS will definitely find it out.
3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages of Breadth-First Search

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is $O(b^d)$. As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.
2. If the solution is farther away from the root, breath first search will consume lot of time.

Depth First Search (DFS) searches deeper into the problem space. Breadth-first search always generates successor of the deepest unexpanded node. It uses last-in first-out stack for keeping the unexpanded nodes. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack.

Algorithm: Depth First Search

1.If the initial state is a goal state, quit and return success.
2.Otherwise, loop until success or failure is signaled.
a) Generate a state, say E, and let it be the successor of the initial state. If there is no successor, signal failure.
b) Call Depth-First Search with E as the initial state.
c) If success is returned, signal success. Otherwise continue in this loop.

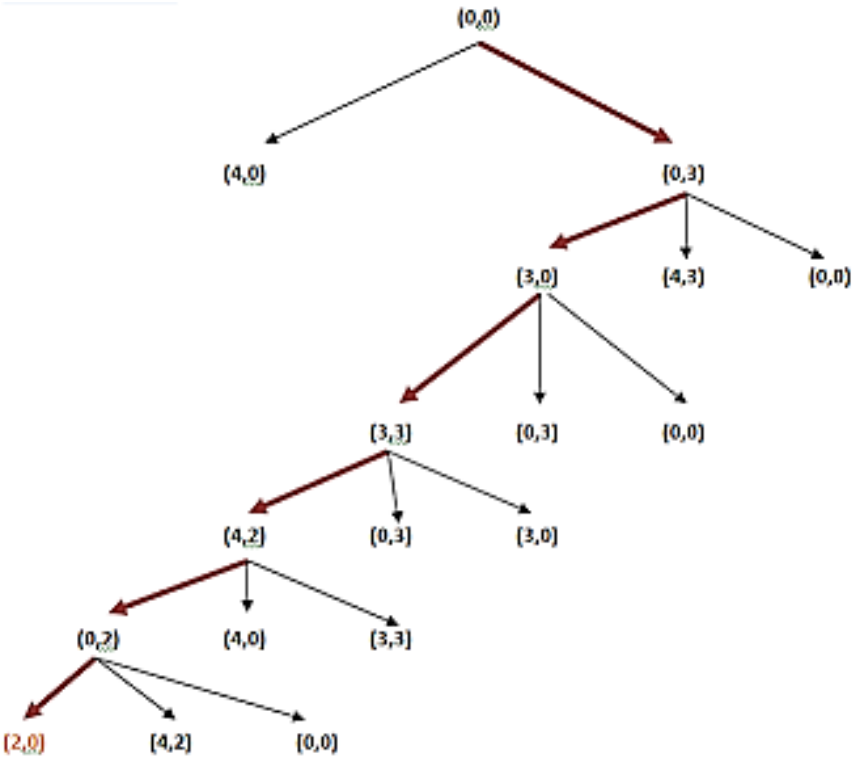
Advantages of Depth-First Search

- The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.

- The time complexity of a depth-first Search to depth d is O(b^d) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.

- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

Solution for Water Jug Problems with Search:



Disadvantages of Depth-First Search

- The disadvantage of Depth-First Search is that there is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree. One solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.

- Depth-First Search is not guaranteed to find the solution.

- And there is no guarantee to find a minimal solution, if more than one solution exists.

1.2.2 Production systems

The production system is a model of computation that can be applied to implement search algorithms and model human problem solving. Such problem solving knowledge can be packed up in the form of little quanta called productions. A production is a rule consisting of a situation recognition part and an action part. A production is a situation-action pair in which the left side is a list of things to watch for and the right side is a list of things to do so. When productions are used in deductive systems, the situation that trigger productions are specified combination of facts. The actions are restricted to being assertion of new facts deduced directly from the triggering combination. Production systems may be called premise conclusion pairs rather than situation action pair.

A production system consists of following components.

(a) A set of production rules, which are of the form A®B. Each rule consists of left hand side constituent that represent the current problem state and a right hand side that represent an output state. A rule is applicable if its left hand side matches with the current problem state.

(b) A database, which contains all the appropriate information for the particular task. Some part of the database may be permanent while some part of this may pertain only to the solution of the current problem.

(c) A control strategy that specifies order in which the rules will be compared to the database of rules and a way of resolving the conflicts that arise when several rules match simultaneously.

(d) A rule applier, which checks the capability of rule by matching the content state with the left hand side of the rule and finds the appropriate rule from database of rules.

1.2.3 Problem characteristics

Heuristic search is a very general method applicable to a large class of problem . It includes a variety of techniques. In order to choose an appropriate method, it is necessary to analyze the problem with respect to the following considerations.

1. **Is the problem decomposable?**

A very large and composite problem can be easily solved if it can be broken into smaller problems and recursion could be used. Suppose we want to solve.

Ex:- ∫ x2 + 3x+sin2x cos 2x dx

This can be done by breaking it into three smaller problems and solving each by applying specific rules. Adding the results the complete solution is obtained.

2. **Can solution steps be ignored or undone?**

Problem fall under three classes ignorable , recoverable and irrecoverable. This classification is with reference to the steps of the solution to a problem. Consider thermo proving. We may later find that it is of no help. We can still proceed further, since nothing is lost by this redundant step. This is an example of ignorable solutions steps.

Now consider the 8 puzzle problem tray and arranged in specified order. While moving from the start state towards goal state, we may make some stupid move and consider theorem proving. We may proceed by first proving lemma. But we may backtrack and undo the unwanted move. This only involves additional steps and the solution steps are recoverable.

Lastly consider the game of chess. If a wrong move is made, it can neither be ignored nor be recovered. The thing to do is to make the best use of current situation and proceed. This is an example of an irrecoverable solution steps.

1. Ignorable problems Ex:- theorem proving

· In which solution steps can be ignored.

2. Recoverable problems Ex:- 8 puzzle

· In which solution steps can be undone

3. Irrecoverable problems Ex:- Chess

· In which solution steps can't be undone

A knowledge of these will help in determining the control structure.

3. **Is the Universal Predictable?**

Problems can be classified into those with certain outcome (eight puzzle and water jug problems) and those with uncertain outcome (playing cards) . in certain – outcome problems, planning could be done to generate a sequence of operators that guarantees to a lead to a solution. Planning helps to avoid unwanted solution steps. For uncertain outcome problems, planning can at best generate a sequence of operators that has a good probability of leading to a solution. The uncertain outcome problems do not guarantee a solution and it is often very expensive since the number of solution and it is often very expensive since the number of solution paths to be explored increases exponentially with the number of points at which the outcome can not be predicted. Thus one of the hardest types of problems to solve is the irrecoverable, uncertain – outcome problems (Ex:- Playing cards).

4. Is good solution absolute or relative?

There are two categories of problems. In one, like the water jug and 8 puzzle problems, we are satisfied with the solution, unmindful of the solution path taken, whereas in the other category not just any solution is acceptable. We want the best, like that of traveling sales man problem, where it is the shortest path. In any – path problems, by heuristic methods we obtain a solution and we do not explore alternatives. For the best-path problems all possible paths are explored using an exhaustive search until the best path is obtained.

5. Is the knowledge base consistent?

In some problems the knowledge base is consistent and in some it is not. For example consider the case when a Boolean expression is evaluated. The knowledge base now contains theorems and laws of Boolean Algebra which are always true. On the contrary consider a knowledge base that contains facts about production and cost. These keep varying with time. Hence many reasoning schemes that work well in consistent domains are not appropriate in inconsistent domains.

Ex: Boolean expression evaluation.

6. What is the role of Knowledge?

Though one could have unlimited computing power, the size of the knowledge base available for solving the problem does matter in arriving at a good solution. Take for example the game of playing chess, just the rues for determining legal moves and some simple control mechanism is sufficient to arrive at a solution. But additional knowledge about good strategy and tactics could help to constrain the search and speed up the execution of the program. The solution would then be realistic.

Consider the case of predicting the political trend. This would require an enormous amount of knowledge even to be able to recognize a solution , leave alone the best.

Ex:- 1. Playing chess 2. News paper understanding

7. Does the task requires interaction with the person.

The problems can again be categorized under two heads.

1. Solitary in which the computer will be given a problem description and will produce an answer, with no intermediate communication and with he demand for an explanation of the reasoning process. Simple theorem proving falls under this category . given the basic rules and laws, the theorem could be proved, if one exists.

Ex:- theorem proving (give basic rules & laws to computer)

2. Conversational, in which there will be intermediate communication between a person and the computer, wither to provide additional assistance to the computer or to provide additional informed information to the user, or both problems such as medical diagnosis fall under this

category, where people will be unwilling to accept the verdict of the program, if they can not follow its reasoning.

Ex:- Problems such as medical diagnosis.

8. **Problem Classification**

Actual problems are examined from the point of view, the task here is examine an input and decide which of a set of known classes.

Ex:- Problems such as medical diagnosis , engineering design.

1.2.4 Production System characteristics

Four classes of production systems:-

1. A monotonic production system

2. A non-monotonic production system

3. A partially commutative production system

4. A commutative production system.

Monotonic production system:- A system in which the application of a rule never prevents the later application of another rule, that could have also been applied at the time the first rule was selected.

Partially commutative production system:-

A production system in which the application of a particular sequence of rules transforms state X into state Y, then any permutation of those rules that is allowable also transforms state x into state Y.

Theorem proving falls under monotonic partially communicative system. Blocks world and 8 puzzle problems like chemical analysis and synthesis come under monotonic, not partially commutative systems. Playing the game of bridge comes under non monotonic, not partially commutative system.

For any problem, several production systems exist. Some will be efficient than others. Though it may seem that there is no relationship between kinds of problems and kinds of production systems, in practice there is a definite relationship.

Partially commutative, monotonic production systems are useful for solving ignorable problems. These systems are important for man implementation standpoint because they can be implemented without the ability to backtrack to previous states, when it is discovered that an incorrect path was followed. Such systems increase the efficiency since it is not necessary to keep track of the changes made in the search process.

Monotonic partially commutative systems are useful for problems in which changes occur but can be reversed and in which the order of operation is not critical (ex: 8 puzzle problem).

Production systems that are not partially commutative are useful for many problems in which irreversible changes occur, such as chemical analysis. When dealing with such systems, the order in which operations are performed is very important and hence correct decisions have to be made at the first time itself.

	monotonic	Non-monotonic
Partially commutative	Theorem proving	Robot navigation
Non-partially commutative	Chemical synthesis	backgammon

1.2.5 Issues in the design of search programs

The following issues are to be considered while designing a search program.

- The direction in which to conduct the search (*forward* versus *backward* reasoning). We can search forward through the state space from the start state to a goal state, or we can search backward from the goal.
- How to select applicable rules *(matching).* Production systems typically spend most of their time looking for rules to apply, so it is critical to have efficient procedures for matching rules against states.
- How to represent each node of the search process (the *knowledge representation problem* and the *frame problem).* For problems like chess, a node can be fully represented by a simple array. In more complex problem solving, however, it is inefficient and/or impossible to represent all of the facts in the world and to determine all of the side effects an action may have.

1.3 Heuristic Approach:

1.3.1 Generate and Test:

The generate and test approach is the simplest of all approaches.

Algorithm:

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 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 I.

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" may be 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. In this form, it is also known as the British Museum algorithm, a reference to a method for finding an object in the British Museum by wandering randomly.' Between the two extremes lies a practical middle ground in which the search process proceeds systematically, but some paths are not considered because they seem unlikely to lead to a solution.

The most straight forward 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 inverse 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 one 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, 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 about the next block. Using this heuristic, many configurations need never be explored and a solution can be found quite quickly.

Unfortunately for problems much harder than this, even heuristic generate-and-test, all by itself, is not a very effective technique. But when combined with other techniques to restrict the space in which to search even further, the technique can be very effective.

For example one early example of a successful Al program is DENDRAL (Lindsay *et at.* 1980]. which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data It uses *a* strategy called *plan-generate-and-test* in which a planning process that uses constraint-satisfaction techniques creates lists of recommended and contraindicated substructures. The generate-and-test procedure then uses those lists so that it can explore only a fairly limited set of structures. Constrained in this way, the generate-and-test procedure has proved highly effective.

This combination of planning, using one problem-solving method *(*in this case. constraint satisfaction) with the use of the plan by another problem-solving method, generate-and-test is an excellent example of the way techniques can be combined to overcome the limitations that each possesses individually. A major weakness of planning is that it often produces somewhat inaccurate solutions since there is no feedback from the world. But by using it only to produce pieces of solutions that will then be exploited in the generate-and-test process, the lack of detailed accuracy becomes unimportant. And, at the same time, the combinatorial problems that arise in simple generate-and-test are avoided by judicious reference to the plans.

1.3.2 Hill Climbing:

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds with only a yes or no. But if the test function is augmented with a heuristic function that provides an estimate of how close a given state is to a goal state, the generate procedure can exploit it as shown in the procedure below. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a

map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

1.3.2.1 Simple Hill Climbing:

The simplest way to implement hill climbing is *as* follows.

Algorithm: Simple Hill Climbing

I. 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 a 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 generate-and-test is the use of an evaluation function as a way to inject task-specific knowledge into the control process. It is the use of such knowledge that makes this a heuristic search method, and it is that same knowledge that gives these methods their power to solve some otherwise intractable problems.

Notice that in this algorithm, a relatively vague question was asked, "Is one state *better* than another?' For the algorithm to work, a precise definition of *better* must be provided. In some cases, it means 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, let's return to the 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 described in the last section. 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.

1.3.2.2 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 he 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.

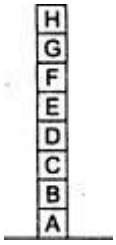
A *ridge* is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the high region compared to the set of available moves and the directions in which they move, makes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

Even with these first-aid measures, hill climbing is not always very effective. It is particularly unsuited to problems where the value of the heuristic function drops off suddenly as you move away from a solution. This is often the case whenever any sort of threshold effect is present. 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 an advantage of being less combinatorial 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 Fig.

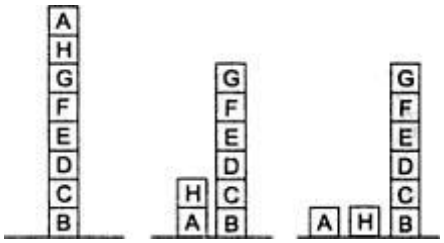


Assume the same operators (i.e.. pick up one block and put it on the table; pick up one block and put it on another one) . Suppose we use the following Heuristic function:

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. These states have the scores:

(a) 4. (b) 4. and (c) 4.



a b c

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.

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.

Unfortunately, it is not always possible to construct such a perfect heuristic function. For example, consider again the problem of driving downtown. The perfect heuristic function would need to have knowledge about one-way and dead-end streets, which, in the case of a strange city, is not always available. And even if perfect knowledge is in principle available, it may not be computationally tractable to use.