Artificial Intelligence

Subject Code: P16CSE2B

Unit V – Short Notes

Note: The following contents of the document are sourced from the reference mentioned at the end of this document.

Game playing:

There were two reasons that games appeared to be a good domain in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position.

The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine. Unfortunately, the second is not true for any but the simplest games. For example, consider chess.

- The average branching factor is around 35.
- In an average game, each player might make 50 moves.
- So in order to examine the complete game tree, we would have to examine 35^{100} positions.

Thus it is clear that a program that simply does a straightforward search of the game tree will not be able to select even its first move during the lifetime of its opponent. Some kind of heuristic search procedure is necessary.

One way of looking at all the search procedures we have discussed is that they are essentially generate-and-test procedures in which the testing is done after varying amounts of work by the generator. At one extreme, the generator generates entire proposed solutions, which the tester then evaluates. At the other extreme, the generator generates individual moves in the search space, each of which is then evaluated by the, tester and the most promising one is chosen. Looked at this way, it is clear that to improve the effectiveness of a search-based problem-solving program two things can be done:

- Improve the generate procedure so that only good moves (or paths) are generated.
- Improve the test procedure so that the best moves (or paths) will be recognized and explored first.

In game-playing programs, it is particularly important that both these things be done. Consider again the problem of playing chess. On the average, there are about 35 legal moves available at each turn. If we use a simple legal-move generator, then the test procedure (which probably uses some combination of search and a heuristic evaluation function) will have to look at each of them. Because the test procedure must look at so many possibilities, it must be fast. So it probably cannot do a very accurate job. Suppose, on the other hand, that instead of a legal-move generator, we use a *plausible-move generator* in which only some small number of promising moves are generated. As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise. (So, for example, it is extremely important in programs that play the game of go [Benson *et al.*, 1979].) With a more selective move generator, the test procedure can afford to spend more time evaluating each of the moves it is given so it can produce a more reliable result. Thus by incorporating heuristic knowledge into both the generator and the tester, the performance of the overall system can be improved.

Of course, in game playing, as in other problem domains, search is not the only available technique. In some games, there are at least some times when more direct techniques are appropriate. For example, in chess, both openings and endgames are often highly stylized, so they are best played by table lookup into a database of stored patterns. To play an entire game then, we need to combine search-oriented and nonsearch-oriented techniques.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached. In the context of game-playing programs, a goal state is one in which we win. Unfortunately, for interesting games such as chess, it is not usually possible, even with a good plausible-move generator, to search until a goal state is found. The depth of the resulting tree (or graph) and its branching factor are too great. In the amount of time available, it is usually possible to search a tree only ten or twenty moves (called *ply* in the game-playing literature) deep. Then, in order to choose the best move, the resulting board positions must be compared to discover which is most advantageous. This is done using a *static evaluation function,* which uses whatever information it has to evaluate individual board positions by estimating how likely they are to lead eventually to a win. Its function is similar to that of the heuristic function *h'* In the A* algorithm: in the absence of complete information, choose the most promising position. Of course, the static evaluation function could simply be applied directly to the positions generated by the proposed moves. But since it is hard to produce a function like this that is very accurate, it is better to apply it as many levels down in the game tree as time permits.

The minimax search procedure:

The *minimax search procedure'is* a depth-first, depth-limited search procedure. It was described briefly in Section 1.3.1. The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. Now we can apply the static evaluation function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent our evaluation of it. The starting position is exactly as good for us as the position generated by the best move we can make next. Here we assume that the static evaluation function returns large values to indicate good situations for us, so our goal is to *maximize* the value of the static evaluation function of the next board position.

An example of this operation is shown in Fig. 12.1. It assumes a static evaluation function that returns values ranging from – 10 to 10, with 10 indicating a win for us, –10 a win for the opponent, and 0 an even match. Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8, since we know we can move to a position with a value of 8.

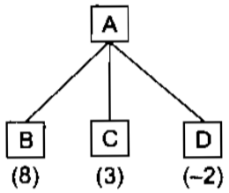

Fig. 12.1 *One-Ply Search*

But since we know that the static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply. This could be very important, for example, in a chess game in which we are in the middle of a piece exchange. After our move, the situation would appear to be very good, but, if we look one move ahead, we will see that one of our pieces also gets captured and so the situation is not as favorable as it seemed. So we would like to look ahead to see what will happen to each of the new game positions at-the next move which will be made by the opponent. Instead of applying the static evaluation function to each of the positions that we just generated, we apply the plausible-move generator, generating a set of successor positions for each position. If we wanted to stop here, at two-ply lookahead, we could apply the static evaluation function to each of these positions, as shown in Fig. 12.2.

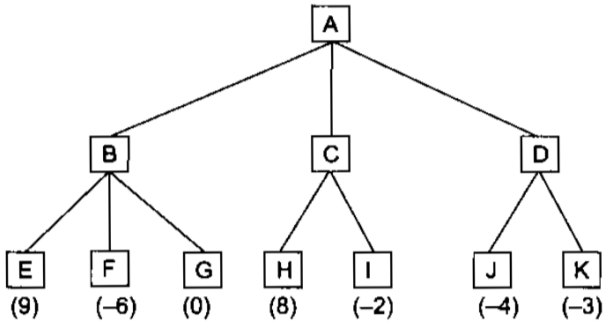

Fig. 12.2 *Two-Ply Search*

But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should be backed up to the next level. Suppose we made move B. Then the opponent must choose among moves E, F, and G. The opponent's goal is to *minimize* the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. This is true even though a possible configuration is that represented by node E, which is very good for us. But since at this level we are not the ones to move, we will not get to choose it. Figure 12.3 shows the result of propagating the new values up the tree. At the level representing the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen.

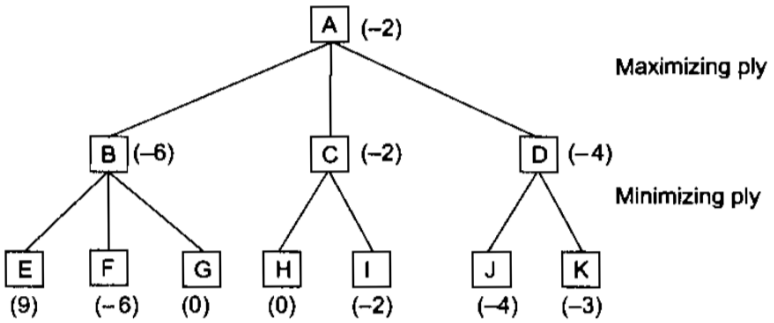

Fig. 12.3 *Backing Up the Values of a Two-Ply Search*

Having described informally the operation of the minimax procedure, we now describe it precisely. It is a straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game being played:

1. MOVEGEN(*Position, Player*)—The plausible-move generator, which returns a list of nodes representing the moves that can be made by *Player* in *Position*. We call the two players PLAYER-ONE and PLAYER-TWO; in a chess program, we might use the names BLACK and WHITE instead.
2. STATIC(*Position, Player*)—The static evaluation function, which returns a num ber representing the goodness of *Position* from the standpoint of *Player*.[2]

As with any recursive program, a critical issue in the design of the MINIMAX procedure is when to stop the recursion and simply call the static evaluation function. There are a variety of factors that may influence this decision. They include:

- Has one side won?
- How many ply have we already explored?
- How promising is this path?
- How much time is left?
- How stable is the configuration?

For the general MINIMAX procedure discussed here, we appeal to a function, DEEP-ENOUGH, which is assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise. Our simple implementation of DEEP-ENOUGH will take two parameters, *Position* and *Depth*. It will ignore its *Position* parameter and simply return TRUE if its *Depth* parameter exceeds a constant cutoff value.

One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results:

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.

We assume that MINIMAX returns a structure containing both results and that we have two functions, VALUE and PATH, that extract the separate components.

Since we define the MINIMAX procedure as a recursive function, we must also specify how it is to be called initially. It takes three parameters, a board position, the current depth of the search, and the player to move. So the initial call to compute the best move from the position CURRENT should be

```
MINIMAX(CURRENT, 0, PLAYER-ONE)
```

if PLAYER-ONE is to move, or

```
MINIMAX(CURRENT, 0, PLAYER-TWO)
```

if PLAYER-TWO is to move.

Algorithm: MINIMAX(Position, Depth, Player)

1. If DEEP-ENOUGH(*Position, Depth*), then return the structure

   VALUE = *STATIC*(*Position, Player*);
   PATH = nil

   This indicates that there is no path from this node and that its value is that determined by the static evaluation function.
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.
   Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the hest score that can be achieved by an element of SUCCESSORS.
   For each element SUCC of SUCCESSORS, do the following:
   (a) Set RESULT-SUCC to
      MINIMAX(SUCC, *Depth* + 1, OPPOSITE(*Player*))
   This recursive call to MINIMAX will actually carry out the exploration of SUCC.
   (b) Set NEW-VALUE to - VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.
   (c) If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
      (i) Set BEST-SCORE to NEW-VALUE.
      (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined hy the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure
   VALUE = BEST-SCORE
   PATH = BEST-PATH

When the initial call to MINIMAX returns, the best move from CURRENT is the first element on PATH. To see how this procedure works, you should trace its execution for the game tree shown in Fig. 12.2.

The MINIMAX procedure just described is very simple. But its performance can be improved significantly with a few refinements. Some of these are described in the next few sections.

Expert System:

Expert systems solve problems that are normally solved by human "experts". To solve expert-level problems, expert systems need access to a substantial domain knowledge base, which must be built as efficiently as possible. They also need to explore one or more reasoning mechanisms to apply their knowledge to the problems they are given. Then they need a mechanism for explaining what they have done to the users who rely on them. One way to look at the expert system is that they represent applied AI in a very broad sense. Expert systems are complex AI programs. The most widely used away of representing domain knowledge in expert systems in as a set of production rules, which are often coupled with a frame system that defines the objects that occur in the rules.

Expert system shells: Since the expert systems were constructed as a set of declarative representations (mostly rules) combined with an interpreter for those representations, it was possible to separate the interpreter from the

domain-specific knowledge and thus to create a system that could be used to construct new expert systems by adding new knowledge corresponding to the new problem domain. The result interpreters are called *shells*.

In order for an expert system to be an effective tool, people must be able to interact with it easily. To facilitate this interaction, the expert system must have the following two capabilities in addition to the ability to perform its underlying task:

- Explain its reasoning. In many of the domains in which expert systems operate, people will not accept results unless they have been convinced of the accuracy of the reasoning process that produced those results. This is particularly true, for example, in medicine, where a doctor must accept ultimate responsibility for a diagnosis, even if that diagnosis was arrived at with considerable help from a program. Thus it is important that the reasoning process used in such programs proceed in understandable steps and that enough meta-knowledge (knowledge about the reasoning process) be available so the explanations of those steps can be generated.
- Acquire new knowledge and modifications of old knowledge. Since expert systems derive their power from the richness of the knowledge bases they exploit, it is extremely important that those knowledge bases be as complete and as accurate as possible. But often there exists no standard codification of that knowledge; rather it exists only inside the heads of human experts. One way to get this knowledge into a program is through interaction with the human expert. Another way is to have the program learn expert behavior from raw data.

Knowledge acquisition:

How are expert systems built? Typically, a knowledge engineer interviews a domain expert to elucidate expert knowledge, which is then translated into rules. After the initial system is built, it must be iteratively refined until it approximates expert-level performance. This process is expensive and time-consuming, so it is worthwhile to look for more automatic ways of constructing expert knowledge bases. While no totally automatic knowledge acquisition systems yet exist, there are many programs that interact with domain experts to extract expert knowledge efficiently. These programs provide support for the following activities:

- Entering knowledge
- Maintaining knowledge base consistency
- Ensuring knowledge base completeness

The most useful knowledge acquisition programs are those that are restricted to a particular problem-solving paradigm, e.g., diagnosis or design. It is important to be able to enumerate the roles that knowledge can play in the problem-solving process. For example, if the paradigm is diagnosis, then the program can structure its knowledge base around symptoms, hypotheses, and causes. It can identify symptoms for which the expert has not yet provided causes. Since one symptom may have multiple causes, the program can ask for knowledge about how to decide when one hypothesis is better than another. If we move to another type of problem-solving, say designing artifacts, then these acquisition strategies no longer apply, and we must look for other ways of profitably interacting with an expert. We now examine two knowledge acquisition systems in detail.

Perception and Action:

Perception involves interpreting sights, sounds, smells, and touch. Action includes the ability to navigate through the world and manipulate objects. In order to build robots that live in the world, we must come to understand these processes. Figure 21.1 shows a design for a complete autonomous robot. Most of AI is concerned only with cognition, the idea being that when intelligent programs are developed, we will simply add sensors and effectors to them. But problems in perception and action are substantial in their own right and are being tackled by researchers in the field of robotics.

In the past, robotics and AI have been largely independent endeavors, and they have developed different techniques to solve different problems. We attempt to characterize the field of robotics at the end of this chapter, but for now, we should note one key difference between AI programs and robots: While AI programs usually operate in computer-simulated worlds, robots must operate in the physical world. As an example, consider making a move in chess. An AI program can search millions of nodes in a game tree without ever having to sense or touch anything in the real world. A complete chess-playing robot, on the other hand, must be capable of grasping pieces, visually interpreting board positions, and carrying on a host of other actions.

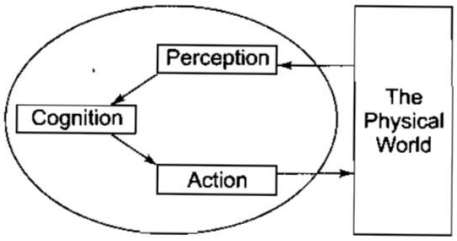

Fig. 21.1 *A Design for an Autonomous Robot*

The distinction between real and simulated worlds has several implications:

- The input to an AI program is symbolic in form, e.g., an 8-puzzle configuration or a typed English sentence. The input to a robot is typically an analog signal, such as a two-dimensional video image or a speech waveform.
- Robots require special hardware for perceiving and affecting the world, while AI programs require only general-purpose computers.
- Robot sensors are inaccurate, and their effectors are limited in precision. There is always some degree of uncertainty about exactly where the robot is located, and where objects and obstacles stand in relation to it. Robot effectors are also limited in precision.
- Many robots must react in real time. A robot fighter plane, for example, cannot afford to search optimally or to stop monitoring the world during a LISP garbage collection.
- The real world is unpredictable, dynamic, and uncertain. A robot cannot hope to maintain a correct and complete description of the world. This means that a robot must consider the trade-off between devising and executing plans. This trade-off has several aspects. For one thing, a robot may not possess enough information about the world for it to do any useful planning. In that case, it must first engage in information-gathering activity. Furthermore, once it begins executing a plan, the robot must continually monitor the results of its actions. If the results are unexpected, then replanning may be necessary. Consider the problem of traveling across town. We might decide to take a bus, but without a bus schedule, it is impossible to complete the plan. So we make a plan for acquiring a schedule and execute it in the world. Now we can plan our route. The bus we want to take may be scheduled to arrive at 5:22 p.m., but the probability of it coming at exactly 5:22 p.m. is actually very small. We should stick to our plan and wait, even if the bus is late. After a while, if the bus still has not come, we must make a new plan.
- Because robots must operate in the real world, searching and backtracking can be costly. Consider the problem of moving furniture into a room. Operating in a simulated world with full information, an AI program can come up with an optimal plan by best-first search. Preconditions of operators can be checked quickly, and if an operator fails to apply, another can be tried. Checking preconditions in the real world, however, can be time-consuming if the robot does not have full information. For example, one operator may require that an object weigh less than fifty pounds. Navigating to the object and applying a force to it may take the robot several minutes. At that rate, it is impossible to traverse and backtrack over a large search space. Worse still, it may be impossible to evaluate a projected arrangement of furniture without actually moving the pieces first.

Perception:

We perceive our environment through many channels: sight, sound, touch, smell, taste. Many animals possess these same perceptual capabilities, and others are able to monitor entirely different channels. Robots, too, can process visual and auditory information, and they can also be equipped with more exotic sensors, such as laser rangefinders, speedometers, and radar.

Two extremely important sensory channels for humans are vision and spoken language. It is through these two faculties that we gather almost all of the knowledge that drives our problem-solving behaviors.

Vision:

A video camera provides a computer with an image represented as a two-dimensional grid of intensity levels. Each grid element, or *pixel,* may store a single bit of information (that is, black/white) or many bits (perhaps a real-valued intensity measure and color information). A visual image is composed of thousands of pixels. What kinds of things might we want to do with such an image? Here are four operations, in order of increasing complexity:

1. Signal Processing—Enhancing the image, either for human consumption or as input to another program.
2. Measurement Analysis—For images containing a single object, determining the two-dimensional extent of the object depicted.
3. Pattern Recognition—For single-object images, classifying the object into a category drawn from a finite set of possibilities.
4. Image Understanding—For images containing many objects, locating the objects in the image, classifying them, and building a three-dimensional model of the scene.

As a result, 2-D images are highly ambiguous. Given a single image, we could construct any number of 3-D worlds that would give rise to the image. For example, consider the ambiguous image of Fig. 21.2. It is impossible to decide what 3-D solid it portrays. In order to determine the most likely interpretation of a scene, we have to apply several types of knowledge.

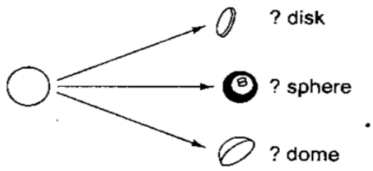

Fig. 21.2 *An Ambiguous Image*

For example, we may invoke knowledge about low-level image features, such as shadows and textures. Figure 21.3 shows how such knowledge can help to disambiguate the image. Having multiple images of the same object can also be useful for recovering 3-D structure. The use of two or more cameras to acquire multiple simultaneous views of an object is called stereo vision. Moving objects (or moving cameras) also supply multiple views. Of course, we must also possess knowledge about how motion affects images that get produced. Still more information can be gathered with a laser rangefinder, a device that returns an array of distance measures much like sonar does. While rangefinders are still somewhat expensive, integration of visual and range data will soon become commonplace. Integrating different sense modalities is called *sensor fusion.* Other image factors we might want to consider include shading, color, and reflectance.

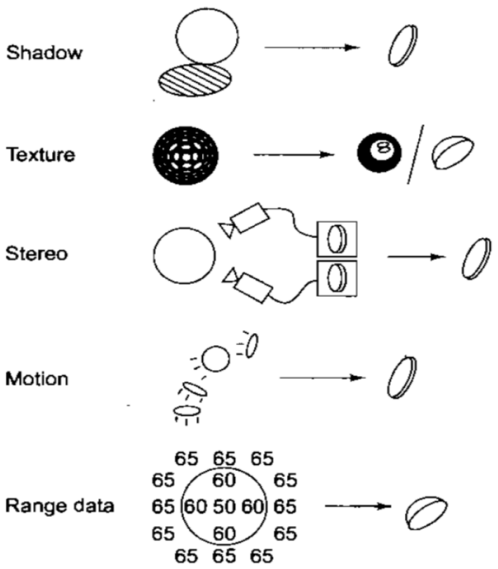

Fig. 21.3 *Using Low-Level Knowledge to Interpret an Image*

High-level knowledge is also important for interpreting visual data. For example, consider the ambiguous object at the center of Fig. 21.4(a). While no low-level image features can tell us what the object is, the object's surroundings provide us with top-down expectations. Expectations are critical for interpreting visual scenes, but resolving expectations can be tricky. Consider the scene shown in Fig. 21.4(b). All objects in this scene are ambiguous; the same shapes might be interpreted elsewhere as an amoeba, logs in a fireplace, and a basketball. As a result, there are no clear-cut top-down expectations. But the preferred interpretations of egg, bacon, and plate reinforce each other mutually, providing the necessary expectations.

Fig. 21.4 *Using High-Level Knowledge ta*

Speech Recognition:

Design issues also provide dimensions along which systems can be compared with one another:

- Speaker Dependence versus Speaker Independence—A speaker-independent system can listen to any speaker and translate the sounds into written text. Speaker independence is hard to achieve because of the wide variations in pitch and accent. It is easier to build a speaker-dependent system, which can be trained on the voice patterns of a single speaker. The system will only work for that one speaker. It can be retrained on another voice, but then it will no longer work for the original speaker.
- Continuous versus Isolated-Word Speech—Interpreting isolated-word speech, in which the speaker pauses between each word, is easier than interpreting continuous speech. This is because boundary effects cause words to be pronounced differently in different contexts. For example, the spoken phrase "could you" contains a *j* sound, and despite the fact it contains two words, there is no empty space between them in the speech wave. The ability to recognize continuous speech is very important, however, since humans have difficulty speaking in isolated words.
- Real Time versus Offline Processing—Highly interactive applications require that a sentence be translated into text as it is being spoken, while in other situations, it is permissihle to spend minutes in computation. Real-time speeds are hard to achieve, especially when higher-level knowledge is involved.
- Large versus Small Vocabulary—Recognizing utterances that are confined to small vocabularies (e.g., 20 words) is easier than working with large vocabularies (e.g., 20,000 words).A small vocabulary helps to limit the number of word candidates for a given speech segment.
- Broad versus Narrow Grammar—An example of a narrow grammar is the one for phone numbers: S → XXX-XXXX, where X is any number between zero and nine: Syntactic and semantic constraints for unrestricted English are much harder to represent, as we saw in Chapter 15. The narrower the grammar is, the smaller the search space for recognition will be.

Action:

Mobility and intelligence seem to have evolved together. Immobile creatures have little use for intelligence, while it is intelligence that puts mobility to effective use. In this section, we investigate the nature of mobility in terms of how robots navigate through the world and manipulate objects.

Navigation:

Navigation means moving around the world: planning routes, reaching desired desti-nations without bumping into things, and so forth. Like vision and speech recognition, this is something humans do fairly easily.

Navigational problems are surprisingly complex. For example, suppose that there are obstacles in the robot's path, as in Fig. 21.6. The problem *of path planning* is to plot a continuous set of points connecting the initial position of the robot to its desired position.

If the robot is so small as to be considered a point, the problem can be solved straightforwardly by constructing a *visibility graph*. Let S be the set consisting of the initial and final positions as well as the vertices of all obstacles. To form the visibility graph, we connect every pair of points in S that are visible from one another, as shown in Fig. 21.7. We can then search the graph (perhaps using the A* algorithm) to find an optimal path for the robot.

Most robots have bulky extent, however, and we must take this into account when we plan paths. Consider the problem shown in Fig. 21.8, where the robot has a pentagonal shape. Fortunately, we can reduce this problem to the previous path-planning problem. The algorithm is as follows: First choose a point P on the surface of the robot, then increase the size of the obstacles so that they cover all points that P cannot enter, because of the physical size and shape of the robot. Now, simply construct and search a visibility graph based on P and the vertices of the new obstacles, as in Fig. 21.9. The basic idea is to reduce the robot to a point P and do path planning in an artificially constructed space, known as *configuration space*, or *c-space* [Lozano-Perez *et al.*, 1984].

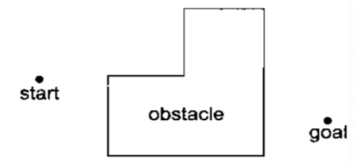

Fig. 21.6 *A Path planning problem*

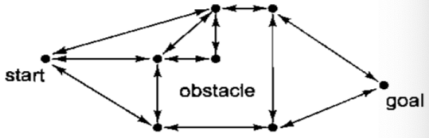

Fig. 21.7 *Constructing a Visibility Graph*

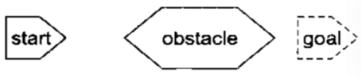

Fig. 21.8 *Another Path Planning Problem*

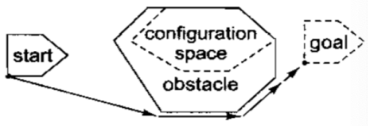

Fig. 21.9 *Constructing Configuration Space Obstacles*

Manipulation:

Robots have found numerous applications in industrial settings. Robot manipulators are able to perform simple repetitive tasks, such as bolting and fitting automobile parts, but these robots are highly task-specific. It is a long-standing goal in robotics to build robots that can be programmed to carry out a wide variety of tasks.

Reference:

1. Rich, E. Kevin Knight. Shivashankar B Nair, 2009. Artificial Intelligence Third Edition, McGraw-Hill Publishing Company Limited, New Delhi.

===End of Unit V===