UNIT II-REPRESENTATION OF KNOWLEDGE (9 hours)

Game playing - Knowledge representation, Knowledge representation using Predicate logic, Introduction to predicate calculus, Resolution, Use of predicate calculus, Knowledge representation using other logic-Structured representation of knowledge.

Prepared by
Vaishnavi Moorthy
Asst Prof- Dept of Cse
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.
Effectiveness of a search based problem solving program

- 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.

- Legal move generator - not promising
- Plausible move generator - small number of promising moves are generated
  - As the number of legal moves increases, promising heuristic is applied.
  - Incorporating knowledge in both tester and generator, the performance of the overall system can be improved
• In the amount of time given—ply in the game playing is limited in a game graph or tree

• Static evaluation function
  □ Eg. Samuels Checkers game
  □ Chess game—piece advantage by Turing

\[ c_1 \times \text{piece advantage} + c_2 \times \text{advancement} + c_3 \times \text{center control} \ldots \]

• Deciding which series of actions are responsible for particular outcome is **Credit assignment problem**

• Eg. A* algorithm—inadequate for chess

• Minmax procedure—works on both standard problem solving trees and on game trees
• Static evaluation function
  • -10 win for the opponent
  • 0 for even match
  • 10 win for us

• Two ply search
DEPTH = 2

\[ B^D = \text{TOTAL NO OF TERMINAL NODES} \]
• States where the game has ended are called terminal states.
• A utility (payoff) function determines the value of terminal states, e.g. win=+10, draw=0, lose=-10.
• In two-player games, assume one is called MAX (tries to maximize utility) and one is called MIN (tries to minimize utility).
• In the search tree, first layer is move by MAX, next layer by MIN, and alternate to terminal states. Each layer in the search is called a ply.
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 number representing the goodness of Position from the standpoint of Player.
Variety of factors affecting the decision

- 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?

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.
Best Move for Current Position

\[
\text{MINIMAX} (\text{CURRENT}, 0, \text{PLAYER-ONE}) \\
\text{if PLAYER-ONE is to move, or} \\
\text{MINIMAX} (\text{CURRENT}, 0, \text{PLAYER-TWO}) \\
\text{if PLAYER-TWO is to move.}
\]

Depth-first and depth-limited search.

• At the player choice, maximize the static evaluation of the next position.

• At the opponent choice, minimize the static evaluation of the next position.
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 best 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 by 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
Alpha Beta Pruning

- Alpha = the value of the best choice we’ve found so far for MAX (highest)
- Beta = the value of the best choice we’ve found so far for MIN (lowest)
- When maximizing, cut off values lower than Alpha
- When minimizing, cut off values greater than Beta
- USE THRESH - used to compute cutoffs
- PASS THRESH - passed to next level
Adding Alpha Beta Cutoffs

An alpha cutoff
Alpha Beta Cutoffs

Maximizing ply

Minimizing ply

Maximizing ply

Minimizing ply
Best Move for Current Position

MINIMAX-A-B(CURRENT,
    0,
    PLAYER-ONE,
    maximum value STATIC can compute,
    minimum value STATIC can compute)
Algorithm: MINIM AX-A-B( Position, Depth, Player, Use-Thresh, Pass-Thresh)

1. If DEEP-ENOUGH(Position, Depth), then return the structure
   \[ \text{VALUE} = \text{STATIC}(\text{Position}, \text{Player}); \]
   \[ \text{PATH} = \text{nil} \]
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, there are no moves to be made; return the same structure that would have
   been returned if DEEP-ENOUGH had returned TRUE.
4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the
   best one. This is done as follows.
   For each element SUCC of SUCCESSORS:
   (a) Set RESULT-SUCC to
       \[ \text{MINIMAX-A-B}(\text{SUCC}, \text{Depth} + 1, \text{OPPOSITE}(<\text{Player}), \]
       \[ -\text{Pass-Thresh}, -\text{Use-Thresh}). \]
   (b) Set NEW-VALUE to −VALUE(RESULT-SUCC).
   (c) If NEW-VALUE > Pass-Thresh, 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 Pass-Thresh to NEW-VALUE.
       (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path
           from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to
           the result of attaching SUCC to the front of PATH(RESULT-SUCC).
   (d) If Pass-Thresh (reflecting the current best value) is not better than Use-Thresh, then we should
       stop examining this branch. But both thresholds and values have been inverted. So if Pass-Thresh
       \[ \geq \text{Use-Thresh}, \]
       then return immediately with the value
       \[ \text{VALUE} = \text{Pass-Thresh}\]
       \[ \text{PATH} = \text{BEST-PATH} \]
5. Return the structure
   \[ \text{VALUE} = \text{Pass-Thresh} \]
   \[ \text{PATH} = \text{BEST-PATH} \]
Alpha-Beta Pruning Example
Alpha-Beta Pruning Example
Alpha-Beta Pruning Example

![Diagram of Alpha-Beta Pruning Example](image)
Alpha-Beta Pruning Example
Alpha-Beta Pruning Example
Additional Requirements

- Waiting for Quiescence

Avoids Horizon effect
• Secondary Search- combating horizon effect (double check)
• Using book moves
• Alternatives to Minimax- predicting weaker move
Iterative deepening

• Ply searches with branch and bound strategy
• Depth first iterative deepening
Algorithm: Depth-First Iterative Deepening

1. Set SEARCH-DEPTH = 1.
2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution path is found, then return it.
3. Otherwise, increment SEARCH-DEPTH by 1 and go to step 2.

DFID is proportional to the number of nodes in that solution path

DFID is slower by a constant factor

Algorithm: Iterative-Deepening-A*

1. Set THRESHOLD = the heuristic evaluation of the start state.
2. Conduct a depth-first search, pruning any branch when its total cost function (g + h’) exceeds THRESHOLD.\(^4\) If a solution path is found during the search, return it.
3. Otherwise, increment THRESHOLD by the minimum amount it was exceeded during the previous step, and then go to Step 2.
References on Specific Games

• Chess
• Checkers
• Go (difficult since average branching factor is very high)
• Backgammon- Neurogammon- automatic learning
• Othello
• Bridge
• Scrabble
• Dominoes
• Go-moku
• Hearts
• poker
MIN MAX PROBLEM

• General method for determining optimal move.
• Generate complete game tree down to terminal states.

\[ \sum = g(f_1, f_2, \ldots, f_n) \rightarrow 1 \]
• where f- features of game g-function and \( \sum \)- static value.

• The g is reduced to a linear squaring polynomial by multiplying a constant

\[ \sum = c_1 f_1, c_2 f_2, \ldots, c_n f_n \rightarrow 2 \]

• Compute utility of each node bottom up from leaves toward root.
• At each MAX node, pick the move with maximum utility.
• At each MIN node, pick the move with minimum utility (assumes opponent always acts correctly to minimize utility).
• When reach the root, optimal move is determined.
ALGORITHM:

function MINIMAX-DECISION(game) returns an operator
   for each op in OPERATORS[game] do
      VALUE[op] MINIMAX-VALUE(APPLY(op, game), game)
   end return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value
   if TERMINAL-TEST[game](state) then
      return UTILITY[game](state)
   else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
   else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
Representations and mappings

- Facts: truths in some relevant world. These are the things we want to represent.
- Representations of facts in some chosen formalism. These are the things we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:
- The *knowledge level*, at which facts (including each agent’s behaviors and current goals) are described.
- The *symbol level*, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.
Spot is a dog.

The fact represented by that English sentence can also be represented in logic as:

\[ \text{dog}(\text{Spot}) \]

Suppose that we also have a logical representation of the fact that all dogs have tails:

\[ \forall x : \text{dog}(x) \rightarrow \text{hastail}(x) \]

Then, using the deductive mechanisms of logic, we may generate the new representation object:

\[ \text{hastail}(\text{Spot}) \]

Using an appropriate backward mapping function, we could then generate the English sentence:

Spot has a tail.

Sample 2

- All dogs have tails
- Every dog has a tail
Representation of Facts

Initial facts \[\rightarrow\] \text{desired real reasoning} \[\rightarrow\] Final facts

\[
\begin{align*}
\text{Internal representation of initial facts} & \overset{\text{forward representation mapping}}{\underset{\text{Operation of program}}{\rightarrow}} \text{Internal representation of initial facts} \\
\end{align*}
\]
Approaches to knowledge representation

Representational Adequacy — the ability to represent all of the kinds of knowledge that are needed in that domain.

- Inferential Adequacy — the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
- Inferential Efficiency — the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mechanisms in the most promising directions.
- Acquisitional Efficiency — the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.
<table>
<thead>
<tr>
<th>Player</th>
<th>Height</th>
<th>Weight</th>
<th>Bats-Throws</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hank Aaron</td>
<td>6-0</td>
<td>180</td>
<td>Right-Right</td>
</tr>
<tr>
<td>Willie Mays</td>
<td>5-10</td>
<td>170</td>
<td>Right-Right</td>
</tr>
<tr>
<td>Babe Ruth</td>
<td>6-2</td>
<td>215</td>
<td>Left-Left</td>
</tr>
<tr>
<td>Ted Williams</td>
<td>6-3</td>
<td>205</td>
<td>Left-Right</td>
</tr>
</tbody>
</table>

`player_info('hank aaron', '6-0', 180, right-right).`

*Simple Relational Knowledge and a sample fact in Prolog*
Inheritable knowledge
Algorithm: Property Inheritance

To retrieve a value $V$ for attribute $A$ of an instance object $O$:

1. Find $O$ in the knowledge base.
2. If there is a value there for the attribute $A$, report that value.
3. Otherwise, see if there is a value for the attribute instance. If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute $A$. If one is found, report it.
5. Otherwise, do until there is no value for the isa attribute or until an answer is found:
   (a) Get the value of the isa attribute and move to that node.
   (b) See if there is a value for the attribute $A$. If there is, report it.
∀x : Ball(x) ∧ Fly(x) ∧ Fair(x) ∧ Infield-Catchable(x) ∧ Occupied-Base(First) ∧ Occupied-Base(Second) ∧ (Outs < 2) ∧ ¬[Line-Drive(x) ∨ Attempted-Bt,(x)] → Infield-Fly(x)

∀x,y : Batter(x) ∧ batted(x, y) ∧ Infield-Fly(y) → Out(x)
Procedural knowledge

If:

ninth inning, and
score is close, and
less than 2 outs, and
first base is vacant, and
batter is better hitter than next batter,

Then:

walk the batter.
Representation Issues

- Are any attributes of objects so basic that they occur in almost every problem domain? If there are, we need to make sure that they are handled appropriately in each of the mechanisms we propose. If such attributes exist, what are they?
- Are there any important relationships that exist among attributes of objects?
- At what level should knowledge be represented? Is there a good set of primitives into which all knowledge can be broken down? Is it helpful to use such primitives?
- How should sets of objects be represented?
- Given a large amount of knowledge stored in a database, how can relevant parts be accessed when they are needed?
Important attributes

• Instance

• Isa

Relationship among attributes

• Inverses

• Existence in an isa hierarchy

• Techniques for reasoning about values

• Single values attributes
Inverses

First approach

\[ \text{team}(\text{Pee-Wee-Reese, Brooklyn-Dodgers}) \]

Second approach

Inheritable Knowledge

- one associated with Pee Wee Reese:
  \[ \text{team} = \text{Brooklyn-Dodgers} \]

- one associated with Brooklyn Dodgers:
  \[ \text{team-members} = \text{Pee-Wee-Reese}, \ldots \]

taken for semantic net and frame based systems accompanied by knowledge acquisition tool
An isa hierarchy of attributes

• Attributes like height - physical attributes
• Generalization-specialization relationships support inheritance
  • Constraints on the values that the attribute can have and mechanisms for computing those values
Techniques for reasoning about values

- Information about the type of the value. For example, the value of *height* must be a number measured in a unit of length.
- Constraints on the value, often stated in terms of related entities. For example, the age of a person cannot be greater than the age of either of that person’s parents.
- Rules for computing the value when it is needed. We showed an example of such a rule in Fig. 4.5 for the *hats* attribute. These rules are called *backward* rules. Such rules have also been called *if-needed rules*.
- Rules that describe actions that should be taken if a value ever becomes known. These rules are called *forward* rules, or sometimes *if-added rules*. 
Single-valued attributes

- Introduce an explicit notation for temporal interval. If two different values are ever asserted for the same temporal interval, signal a contradiction automatically.
- Assume that the only temporal interval that is of interest is now. So if a new value is asserted, replace the old value.
- Provide no explicit support. Logic-based systems are in this category. But in these systems, knowledge-base builders can add axioms that state that if an attribute has one value then it is known not to have all other values.
Choosing the granularity of representation

John spotted Sue.

We could represent this as

$$spotted(agent(John),\text{ object(Sue)})$$

Such a representation would make it easy to answer questions such as:

Who spotted Sue?

But now suppose we want to know:

Did John see Sue?

The obvious answer is “yes,” but given only the one fact we have, we cannot discover that answer. We could, of course, add other facts, such as

$$spotted(x, y) \rightarrow saw(x, y)$$

We could then infer the answer to the question.

An alternative solution to this problem is to represent the fact that spotting is really a special type of seeing explicitly in the representation of the fact. We might write something such as

$$saw(agent(John),\text{ object(Sue),}\text{ timespan(briefly)})$$
"John punched Mary."

(a)

"Mary punched John."

(b)

Redundant Representations
Mary is Sue’s cousin

- $\text{Mary} = \text{daughter(}\text{brother(}\text{mother(Sue))}\text{)}$
- $\text{Mary} = \text{daughter(}\text{sister(}\text{mother(Sue))}\text{)}$
- $\text{Mary} = \text{daughter(}\text{brother(}\text{father(Sue))}\text{)}$
- $\text{Mary} = \text{daughter(}\text{sister(}\text{father(Sue))}\text{)}$

Another representation

$\text{Mary} = \text{child(}\text{sibling(}\text{parent(Sue))}\text{)}$
In less well-structured domains, even more problems arise. For example, given just the fact
John broke the window.

a program would not be able to decide if John's actions consisted of the primitive sequence:

1. Pick up a hard object.
2. Hurl the object through the window.

or the sequence:

1. Pick up a hard object.
2. Hold onto the object while causing it to crash into the window.

or the single action:

1. Cause hand (or foot) to move fast and crash into the window.

or the single action:

1. Shut the window so hard that the glass breaks.
The phrase *Evening Star* names a certain large physical object of spherical form, which is hurtling through space some scores of millions of miles from here. The phrase *Morning Star* names the same thing, as was probably first established by some observant Babylonian. But the two phrases cannot be regarded as having the same meaning; otherwise that Babylonian could have dispensed with his observations and contented himself with reflecting on the meaning of his words. The meanings, then, being different from one another, must be other than the named object, which is one and the same in both cases.
Representation of sets of objects

• Extensional definition- list all members
• Intentional definition- needed members

Evaluate the sentence — true or false

\[ \{x : \text{sun-planet}(x) \land \text{human-inhabited}(x)\} \]

One extensional can have many intentional definitions

\[
\{x : \text{sun-planet}(x) \land \text{nth-farthest-fmm-sun}(x, 3)\} \\
\{x : \text{sun-planet}(x) \land \text{nth-biggest}(x, 5)\}
\]
Finding the right structures as needed

• How to perform an initial selection of the most appropriate structure.
• How to fill in appropriate details from the current situation.
• How to find a better structure if the one chosen initially turns out not to be appropriate.
• What to do if none of the available structures is appropriate.
• When to create and remember a new structure.

John went to Steak and Ale last night. He ordered a large rare steak, paid his bill, and left.

and answer “yes” to the question

Did John eat dinner last night?
• Selecting an initial structure
  – Indexing the sentence with significant English words
  – Pointer to all structures- prospective structures
  – On major clue in the problem description
• Revising the choice when necessary
  • Select the fragments of the current structure- candidate structure- if matches then preserve it
  • If failure- you can consider the related information and make excuses
  • Use stroke lines to specify the direction (new)
  • Isa (upward direction)- no conflict- use structure to provide knowledge representation else new structure