UNIT-5

Module 2

Game Playing:

Game Playing

- Charles Babbage, the nineteenth-century computer architect thought about programming his analytical engine to play chess and later of building a machine to play tic-tac-toe.
- There are two reasons that games appeared to be a good domain.
    1. They provide a structured task in which it is very easy to measure success or failure.
    2. They are easily solvable by straightforward search from the starting state to a winning position.
- The first is true is for all games bust the second is not true for all, except simplest games.
- For example, consider chess.
- The average branching factor is around 35. In an average game, each player might make 50.
- So in order to examine the complete game tree, we would have to examine 35^{100}
- Thus it is clear that a simple search is not able to select even its first move during the lifetime of its opponent.
- It is clear that to improve the effectiveness of a search based problem-solving program two things can do.
    1. Improve the generate procedure so that only good moves generated.
    2. Improve the test procedure so that the best move will recognize and explored first.
- If we use legal-move generator then the test procedure will have to look at each of them because the test procedure must look at so many possibilities, it must be fast.
- Instead of the legal-move generator, we can use plausible-move generator in which only some small numbers of promising moves generated.
- As the number of lawyers available moves increases, it becomes increasingly important in applying heuristics to select only those moves that seem more promising.
- The performance of the overall system can improve by adding heuristic knowledge into both the generator and the tester.
- In game playing, a goal state is one in which we win but the game like chess. It is not possible. Even we have good plausible move generator.
- The depth of the resulting tree or graph and its branching factor is too great.
- It is possible to search tree only ten or twenty moves deep then in order to choose the best move. The resulting board positions must compare to discover which is most advantageous.
- This is done using static evolution function, which uses whatever information it has to evaluate individual board position 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.

MINIMAX Search Procedure

- The minimax search is a depth-first and depth limited procedure.
- 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 evolution 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 evolution of it.
- Here we assume that static evolution function returns larger 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.
- The opponents' goal is to minimize the value of the static evaluation function.
- **The alternation of maximizing and minimizing at alternate ply when evaluations are to be pushed back up corresponds to the opposing strategies of the two players is called MINIMAX.**
- It is the recursive procedure that depends on two procedures
    - MOVEGEN(position, player)— The plausible-move generator, which returns a list of nodes representing the moves that can make by Player in Position.
    - STATIC(position, player)– static evaluation function, which returns a number representing the goodness of Position from the standpoint of Player.
- With any recursive program, we need to decide when recursive procedure should stop.
- There are the variety of factors that may influence the decision they are,
    - Has one side won?
    - How many plies have we already explored? Or how much time is left?
    - How stable is the configuration?
- We use DEEP-ENOUGH which assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise.
- It takes two parameters, position, and depth, it will ignore its position parameter and simply return TRUE if its depth parameter exceeds a constant cut off 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, actually needed.
- We assume that MINIMAX returns a structure containing both results and we have two functions, VALUE and PATH that extract the separate components.
- Initially, It takes three parameters, a board position, the current depth of the search, and the player to move,
    - MINIMAX(current,0,player-one) If player –one is to move
    - MINIMAX(current,0,player-two) If player –two is to move

Adding alpha-beta cutoffs

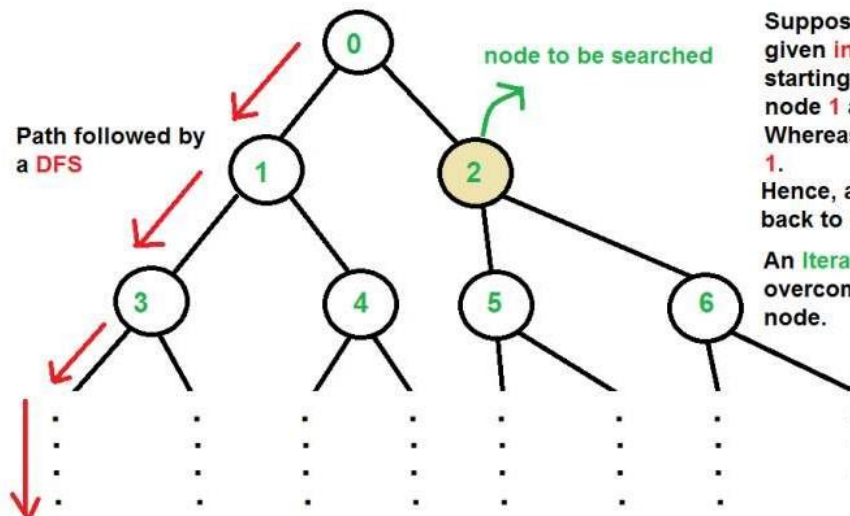
- Minimax procedure is a depth-first process. One path is explored as far as time allows, the static evolution function is applied to the game positions at the last step of the path.
- The efficiency of the depth-first search can improve by branch and bound technique in which partial solutions that clearly worse than known solutions can abandon early.

- It is necessary to modify the branch and bound strategy to include two bounds, one for each of the players.
- This modified strategy called alpha-beta pruning.
- It requires maintaining of two threshold values, one representing a lower bound on that a maximizing node may ultimately assign (we call this alpha).
- And another representing an upper bound on the value that a minimizing node may assign (this we call beta).
- Each level must receive both the values, one to use and one to pass down to the next level to use.
- The MINIMAX procedure as it stands does not need to treat maximizing and minimizing levels differently. Since it simply negates evaluation each time it changes levels.
- Instead of referring to alpha and beta, MINIMAX uses two values, USE-THRESH and PASSTHRESH.
- USE-THRESH used to compute cutoffs. PASS-THRESH passed to next level as its USETHRESH.
- USE-THRESH must also pass to the next level, but it will pass as PASS-THRESH so that it can be passed to the third level down as USE-THRESH again, and so forth.
- Just as values had to negate each time they passed across levels.
- Still, there is no difference between the code required at maximizing levels and that required at minimizing levels.
- PASS-THRESH should always the maximum of the value it inherits from above and the best move found at its level.
- If PASS-THRESH updated the new value should propagate both down to lower levels. And back up to higher ones so that it always reflects the best move found anywhere in the tree.
- The MINIMAX-A-B requires five arguments, position, depth, player, Use-thresh, and passThresh.
- MINIMAX-A-B(current,0,player-one,maximum value static can compute, minimum value static can compute).

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

There are two common ways to traverse a graph, BFS and DFS. Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to following reasons.

1. **DFS** first traverses nodes going through one adjacent of root, then next adjacent. The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges).



2. **BFS** goes level by level, but requires more space. The space required by DFS is O(d) where d is depth of tree, but space required by BFS is O(n) where n is number of nodes in tree (Why? Note that the last level of tree can have around n/2 nodes and second last level n/4 nodes and in BFS we need to have every level one by one in queue).

IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).

How does IDDFS work?

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

Algorithm:

```
// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
   for limit from 0 to max_depth

     if DLS(src,
        target,
        limit) ==
        true
        return
        true

   return false

bool DLS(src, target, limit)
   if (src == target)
      return true;
```

```
// If reached the maximum depth,
// stop recursing.
if (limit <= 0)
    return false;

foreach adjacent i of src

    if DLS(i, target, limit?1)
        return true

return false
```

An important thing to note is, we visit top level nodes multiple times. The last (or max depth) level is visited once, second last level is visited twice, and so on. It may seem expensive, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level. So it does not matter much if the upper levels are visited multiple times.

Planning

Blocks World Problem

In order to compare the variety of methods of planning, we should find it useful to look at all of them in a single domain that is complex enough that the need for each of the mechanisms is apparent yet simple enough that easy-to-follow examples can be found.

- There is a flat surface on which blocks can be placed.
- There are a number of square blocks, all the same size.
- They can be stacked one upon the other.
- There is robot arm that can manipulate the blocks.

Actions of the robot arm

1. UNSTACK(A, B): Pick up block A from its current position on block B.
2. STACK(A, B): Place block A on block B.
3. PICKUP(A): Pick up block A from the table and hold it.
4. PUTDOWN(A): Put block A down on the table. Notice that the robot arm can hold only one block at a time.

Predicates

- In order to specify both the conditions under which an operation may be performed and the results of performing it, we need the following predicates:
    1. ON(A, B): Block A is on Block B.
    2. ONTABLES(A): Block A is on the table.
    3. CLEAR(A): There is nothing on the top of Block A.
    4. HOLDING(A): The arm is holding Block A.
    5. ARMEMPTY: The arm is holding nothing.

Robot problem-solving systems (STRIPS)

- List of new predicates that the operator causes to become true is ADD List
- Moreover, List of old predicates that the operator causes to become false is DELETE List

- PRECONDITIONS list contains those predicates that must be true for the operator to be applied.

STRIPS style operators for BLOCKs World

STACK(x, y)
P:
CLEAR
(y)^HO
LDING(
x) D:
CLEAR
(y)^HO
LDING(
x) A:
ARME
MPTY^
ON(x,
y)
UNSTA
CK(x,
y)
PICKUP(x)
P: CLEAR(x) ^
ONTABLE(x)
^ARMEMPTY D:
ONTABLE(x) ^
ARMEMPTY

A
:

H
O
L
D
I
N
G
(
x
)

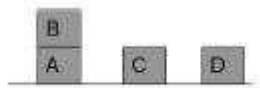
P
U
T
D
O
W
N
(
x
)

Goal Stack Planning

To start with goal stack is simply:

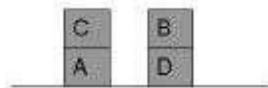
- ON(C,A)^ON(B,D)^ONTABLE(A)^ONTABLE(D)

This problem is separate into four sub-problems, one for each component of the goal. Two of the sub-problems ONTABLE(A) and ONTABLE(D) are already true in the initial state.



Start:

ON(B,A)^ONTABLE(A) ^ ONTABLE(C) ^ONTABLE(D) ^ARMEMPTY



Goal: ON(C,A)^ON(B,D)^ ONTABLE(A)^ONTABLE(D)

Alternative 1: Goal Stack:

- ON(C,A)
- ON(B,D)
- ON(C,

A)^ON(B,D)^
OTAD

Alternative 2:
Goal stack:

- ON(B,D)
- ON(C,A)
- ON(C,A)^ON(B,D)^OTAD

Exploring Operators

- Pursuing alternative 1, we check for operators that could cause ON(C, A)
- Out of the 4 operators, there is only one STACK. So it yields:
    - STACK(C,A)
    - ON(B,D)
    - ON(C,A)^ON(B,D)^OTAD
- Preconditions for STACK(C, A) should be satisfied, we must establish them as sub-goals:
    - CLEAR(A)
    - HOLDING(C)
    - CLEAR(A)^HOLDING(C)
    - STACK(C,A) o ON(B,D)
    - ON(C,A)^ON(B,D)^OTAD
- Here we exploit the Heuristic that if HOLDING is one of the several goals to be achieved at once, it should be tackled last.

Goal stack Planning

- Next, we see if CLEAR(A) is true. It is not. The only operator that could make it true is UNSTACK(B, A). Also, This produces the goal stack:
    - ON(B, A)
    - CLEAR(B)
    - ON(B,A)^CLEAR(B)^ARMEMPTY
    - UNSTACK(B, A)
    - HOLDING(C)

- CLEAR(A)^HOLDING(C)
  - STACK(C,A)
  - ON(B,D)
  - ON(C,A)^ON(B,D)^OTAD
- We see that we can pop predicates on the stack till we reach HOLDING(C) for which we need to find a suitable operator.
- Moreover, The operators that might make HOLDING(C) true: PICKUP(C) and UNSTACK(C, x). Without looking ahead, since we cannot tell which of these operators is appropriate. Also, we create two branches of the search tree corresponding to the following goal stacks:

ALT1:	ALT2:
ONTABLE(C)	ON(C, x)
CLEAR(C)	CLEAR(C)
ARMEMPTY	ARMEMPTY
ONTABLE(C) ^CLEAR(C)^ARMEMPTY	ON(C,x)^CLEAR(C)^ARMEM PTY
PICKUP(C)	UNSTACK(C,x)
CLEAR(A)^HOLDING(C)	CLEAR(A)^HOLDING(C)
STACK(C,A)	STACK(C,A)
ON(B,D)	ON(B,D)
ON(C,A)^ON(B,D)^OTAD	ON(C,A)^ON(B,D)^OTAD

Complete plan

1. UNSTACK(C, A)
2. PUTDOWN(C)
3. PICKUP(A)
4. STACK(A, B)
5. UNSTACK(A, B)
6. PUTDOWN(A)
7. PICKUP(B)
8. STACK(B, C)
9. PICKUP(A)
10. STACK(A,B)

Hierarchical Planning

- In order to solve hard problems, a problem solver may have to generate long plans.
- It is important to be able to eliminate some of the details of the problem until a solution that addresses the main issues is found.
- Then an attempt can make to fill in the appropriate details.
- Early attempts to do this involved the use of macro operators, in which larger operators were built from smaller ones.
- In this approach, no details eliminated from actual descriptions of the operators.

ABSTRIPS

A better approach developed in ABSTRIPS systems which actually planned in a hierarchy of abstraction spaces, in each of which preconditions at a lower level of abstraction ignored.

ABSTRIPS approach is as follows:

- First solve the problem completely, considering only preconditions whose criticality value is the highest possible.
- These values reflect the expected difficulty of satisfying the precondition.

- To do this, do exactly what STRIPS did, but simply ignore the preconditions of lower than peak criticality.
- Once this done, use the constructed plan as the outline of a complete plan and consider preconditions at the next-lowest criticality level.
- Augment the plan with operators that satisfy those preconditions.
- Because this approach explores entire plans at one level of detail before it looks at the lower-level details of any one of them, it has called length-first approach.

The assignment of appropriate criticality value is crucial to the success of this hierarchical planning method.
Those preconditions that no operator can satisfy are clearly the most critical.
Example, solving a problem of moving the robot, for applying an operator, PUSH-THROUGH DOOR, the precondition that there exist a door big enough for the robot to get through is of high criticality since there is nothing we can do about it if it is not true.