ARTIFICIAL INTELLIGENCE

CHAPTER – 3 Solving Problems by Searching

Problem solving agents decide what to do by finding sequences of actions that lead to desirable states.

Uninformed algorithms give no information about the problem other than its definition. Intelligent agents maximize their performance measure by adopting a **goal** and aiming at satisfying it.

Goal formulation based on the current situation and the agent's performance is the first step in problem solving.

Problem formulation is the process of deciding what actions and states to consider given a goal.

An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value and then choosing the best sequence. This process of looking for such a sequence is called **search**.

A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends actions that can be carried out. This is called the execution phase.

Problem solving agent:

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

This figure assumes that environment is static, observable, discrete and deterministic.

Problem can be defined by four components:

a) Initial state

b) Possible actions available to the agent: using successor function

Given a particular state x, SUCCESSOR-FN(x) returns a set if <action, successor> ordered pairs.

Initial state and successor function defines state space of the problem (set of all states reachable from initial state). The state space forms a graph in which the nodes are states and the arcs between nodes are actions. A **path** in the state space is sequence of states connected by sequence of actions.

c) Goal test determines whether a given state is a goal state

d) Path cost assigns numeric cost to each path (sum of costs of individual actions along the path)

Step cost of taking action a to go from x to y: c(x, a, y)

A solution to a problem is a path from the initial state to a goal state. Solution quality is measured by path cost function and an optimal solution has lowest path cost among all solutions.

Abstraction – process of removing detail from representation.

Example problems:

1. Toy problem –

   Vacuum cleaner:

   states initial state successor function goal test path cost
2. Real world problem –

   Travelling sales person problem

Searching for solutions:

Search techniques use explicit search tree that is generated by initial state and successor function that together define the state space. Root of search tree is **search node** corresponding to initial state. The steps to be followed are:

a) Check whether it is a goal state.
b) If it is not the goal state, expand current state by applying successor function to current state thereby generating new set of states.
c) Continue choosing, testing and expanding until either a solution is found or there are no more states to be expanded. Choice of which state to expand depends on search strategy.

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
```

Node is a book keeping data structure with five components:

a) state – state in state space to which node corresponds

b) parent_node – node in search tree that generated this node

c) action – action applied to parent to generate the node

d) path_cost – cost denoted by g(n) from initial state to node indicated by parent pointers

e) depth – number of steps along the path from the initial state

Collection of nodes that have been generated but not yet expanded is called fringe. Each element of fringe is called leaf node (no successors)

Measuring problem solving performance –

a) Completeness: does a solution exist? b) Optimality: is this an optimal solution?

c) Time complexity: how long does it take to find a solution?

d) Space complexity: amount of memory needed to perform a search?

General tree search algorithm-

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[ problem]), fringe)
   loop do
      if EMPTY?( fringe) then return failure
      node ← REMOVE-FIRST( fringe)
      if GOAL-TEST[ problem] applied to STATE[node] succeeds
         then return SOLUTION(node)
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
   successors ← the empty set
   for each ⟨action, result⟩ in SUCCESSOR-FN[ problem](STATE[node]) do
      s ← a new NODE
      STATE[s] ← result
      PARENT-NODE[s] ← node
      ACTION[s] ← action
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
   return successors
```

In AI, graph is represented implicitly by initial state and successor function and is frequently finite; complexity is expressed in three quantities-

B – Branching factor or maximum number of successors of any node; D – Depth of shallowest goal node; m- Maximum length of any path in state space

For finding effectiveness of search algorithm, use

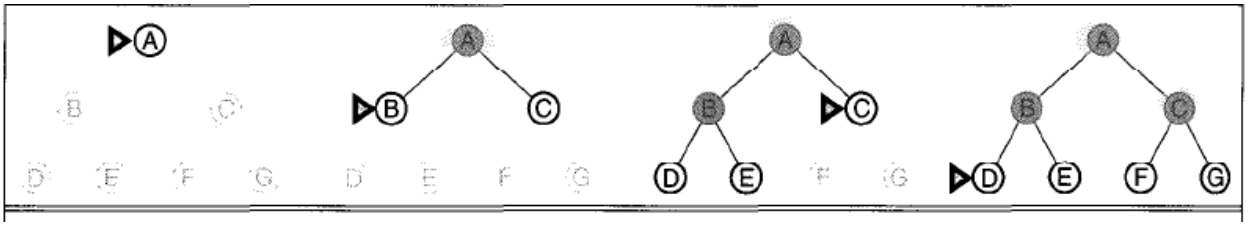
a) Search cost – depends on time complexity
b) Total cost – combines search cost and path cost of solution found

Uninformed search strategies (also called blind search) –

No additional information about states beyond that provided in the problem definition.

Breadth first search: Root node is expanded first, and then all the successors of root node are expanded next and so on.

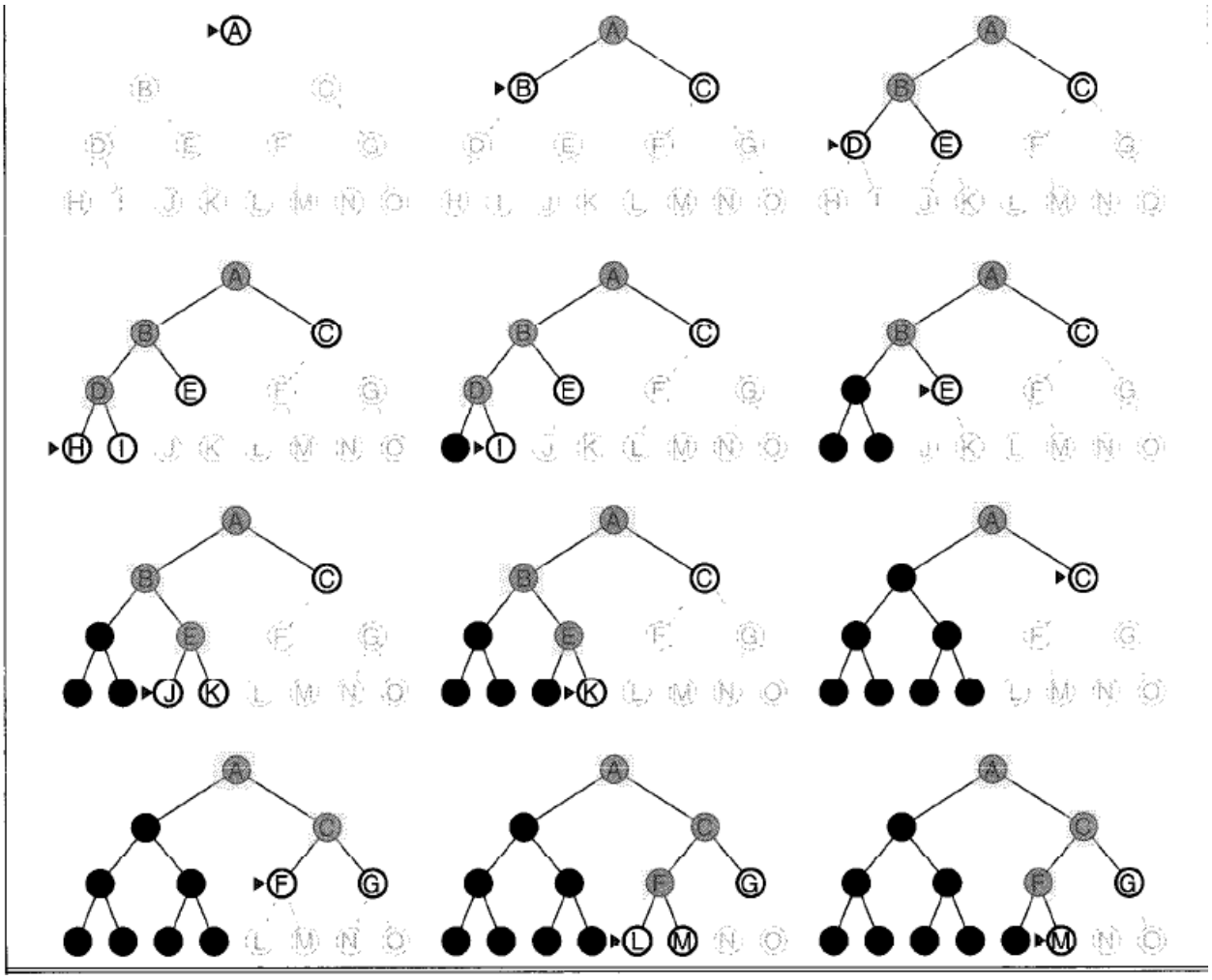
a) Nodes visited first will be expanded first using FIFO approach
b) FIFO puts all newly generated successors at the end of queue which means shallow nodes are expanded first before deeper nodes.



Depth first search: Expands deepest node in current fringe of search tree, search proceeds immediately to the deepest level of search tree where nodes have no successors. As those nodes are expanded they are dropped from the fringe, then the search backs up to the next shallowest node. This is implemented with LIFO queue also known as stack.

Backtracking search: uses less memory compared to DFS. Only one successor is generated at a time rather than all successors, each partially node expanded node remembers which successor to generate next.

Drawback of DFS - gets stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree.



CHAPTER – 4 Informed Search and Exploration

Informed search strategy uses problem specific knowledge thus finding solution more efficiently.

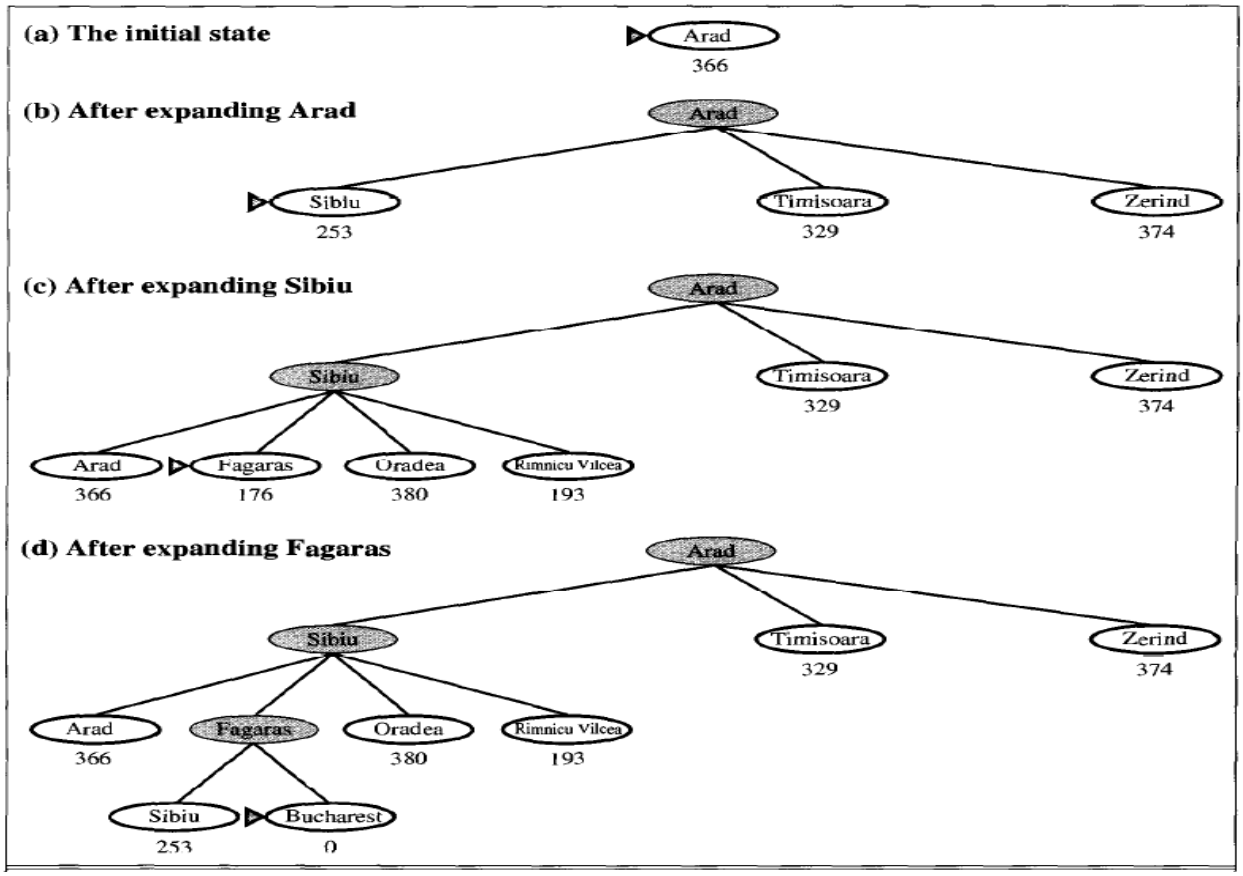
Best first search – node is selected for expansion based on evaluation function f(n).Node with lowest evaluation is selected for expansion. This is implemented via a priority queue. Choose a node that appears to be best according to evaluation function.

Key component of best first search algorithm is heuristic function denoted by h(n).

h(n) – estimated cost of cheapest path from node n to goal node.

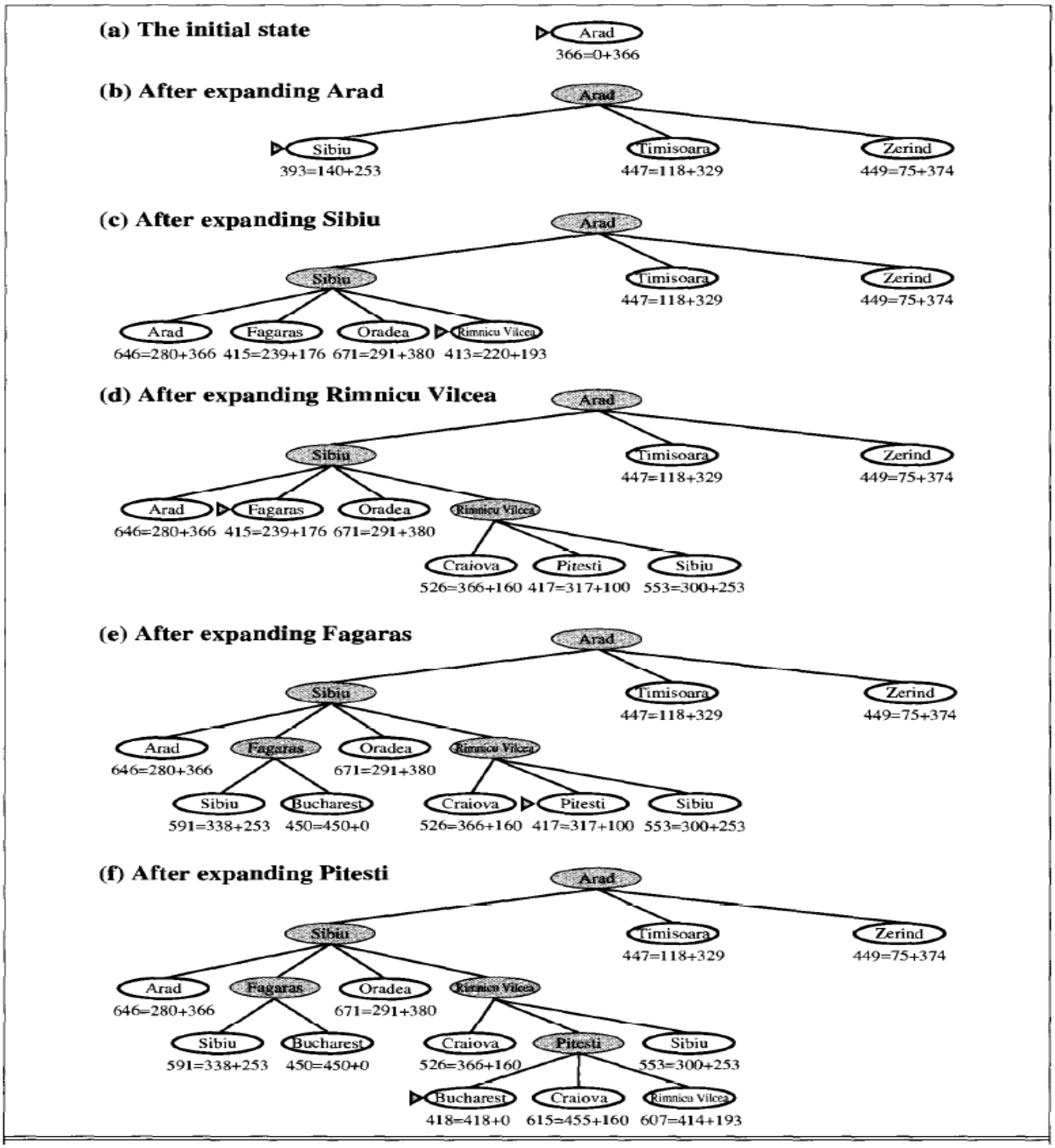
2 ways to use heuristic information –

a) Greedy best first search : tries to expand node that is closest to goal , evaluates nodes by using heuristic function f(n) = h(n)
Algorithm is called greedy because at each step it tries to get close to the goal as it can. Resembles DFS as it prefers to allow single path all the way to the goal but will back up when it hits a dead end.
Disadvantage – not optimal and incomplete.



b) A* search – evaluates nodes by combining g(n) and h(n)
g(n) – cost from start node to node n
h(n) – cost to get from node n to goal.

f(n) = g(n) + h(n) where f(n) is the estimated cost of the cheapest solution through n.
A* is optimal if h(n) is an admissible heuristic provided h(n) never over estimates the cost to reach the goal.



CHAPTER – 6 Adversarial Search

Multi agent environments – agent has to consider actions of other agents

Co – operative environment: taxi driving Competitive environment: playing chess

Unpredictability in competitive multi agent environment gives rise to adversarial search problems known as games. Impact of each agent is significant on the others. In AI, games are deterministic, turn taking, two players, zero sum games of perfect information whose actions must alternate and in which the utility value at end of game are equal and opposite.

Pruning allows us to ignore portions of search tree that make no difference to the final choice.

Heuristic evaluation functions allow approximating the true utility of a state without doing a complete search.

Optimal decisions in games –

A game can be defined as a kind of search problem with the following components –

a) Initial state b) Successor function c) Terminal test d) Utility function

Given a game tree, the optimal strategy can be determined by examining the minimax value of each node which is written as MINIMAX – VALUE (n). The minimax value of a node is the utility (for MAX) of being in the corresponding state assuming that both players play optimally from there to the end of the game.

MAX will prefer to move to a state of maximum value whereas MIN prefers a state of minimum value.

Minimax algorithm –

Computes minimax decision from current state. It uses a simple recursive computation of the minimax values of each successor state directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree and then the minimax values are backed up through the tree as the recursion unwinds.

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$

Optimal decisions in multiplayer games –

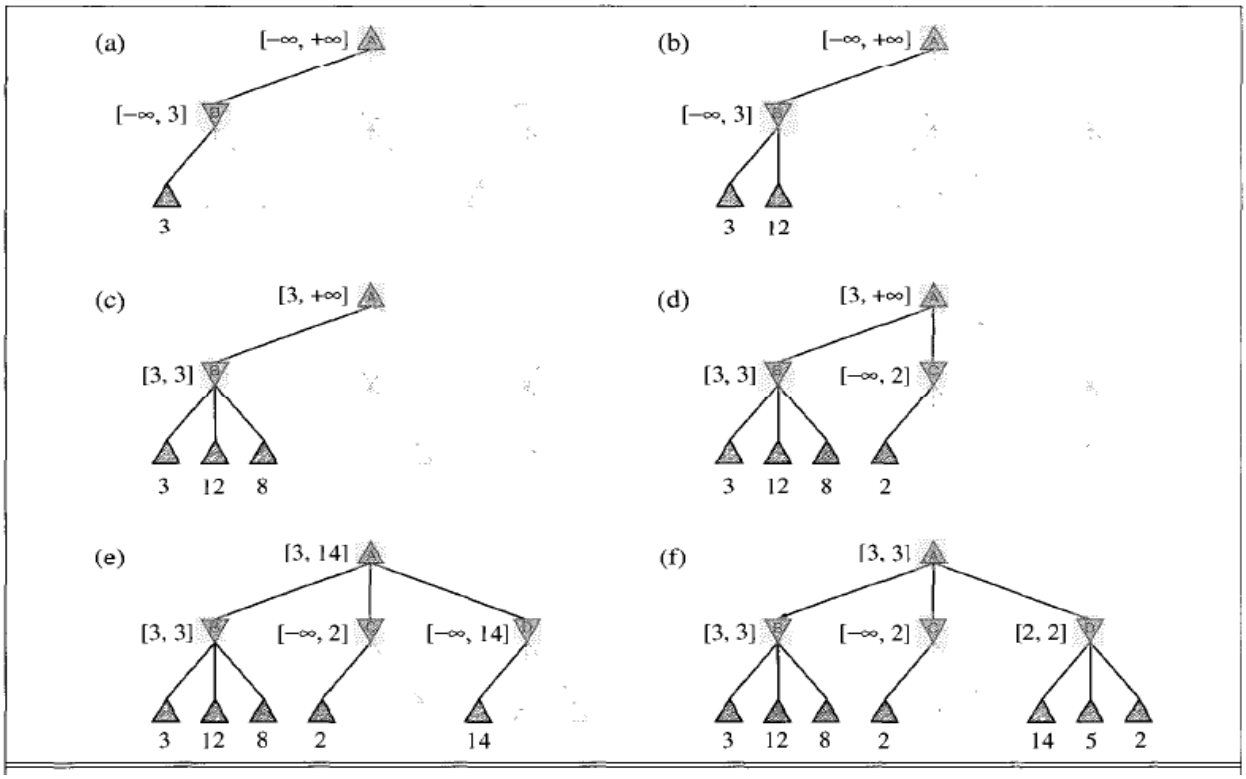
Extending the idea of minimax to multiplayer games. First replace the single value for each node with a vector of values. For terminal states, this vector gives the utility of the state from each player's view point. This can be achieved by having the utility function return a vector of utilities.

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```



Alpha beta pruning –

Compute correct minimax decision by pruning which eliminates large parts of tree from consideration. When applied to a standard minimax tree, it returns the same as minimax would but prunes away branches that cannot possibly influence the final decision. Technique applied is DFS and can be applied to tress of any depth.

Alpha beta pruning gets its name from 2 parameters that describe bounds on the backed – up values that appear anywhere along the path:

Alpha – value of best (highest value) choice found so far at any choice point along path for MAX

Beta – value of best (lowest value) choice found so far at any choice point along path for MIN

Alpha – beta search updates the value of alpha and beta as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than current alpha or beta value for max or min respectively.

Disadvantage – alpha – beta still has to search all the way to terminal states for at least a portion of the search space.

In games, repeated states occur frequently because of transposition – different permutations of the move sequence that end up in the same position.

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Imperfect Real Time Decisions –

Minimax algorithm generates entire game search space whereas alpha beta algorithm allows us to prune large parts of it. Hence programs should cut off the search earlier and apply a heuristic evaluation function to states in the search turning non terminal nodes into terminal leaves. This alters alpha – beta pruning in 2 ways: utility function is replaced by a heuristic evaluation function called EVAL which gives an estimate of the position's utility and the terminal test is replaced by a cut-off test that decides when to apply EVAL.

➔ *Evaluation functions:*

An evaluation that returns an estimate of the expected utility of the game from a given position. First, the evaluation function should order the terminal states in the same way as the true utility function. Second, the computation must not take too long. Third, for non terminal states, the evaluation function should be strongly correlated with the chances of winning.

Most evaluation functions work by calculating various features of the state. These features taken together define various categories or equivalence classes of states – the states in each category have the same values for all the features.

a) Expected value b
b) Material value
c) Weighted linear function

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

➔ *Cutting off search:*

Here we modify alpha – beta search so that it will call the heuristic EVAL function when it is appropriate to cut off the search.

If CUTOFF-TEST (state, depth) then return EVAL (state)