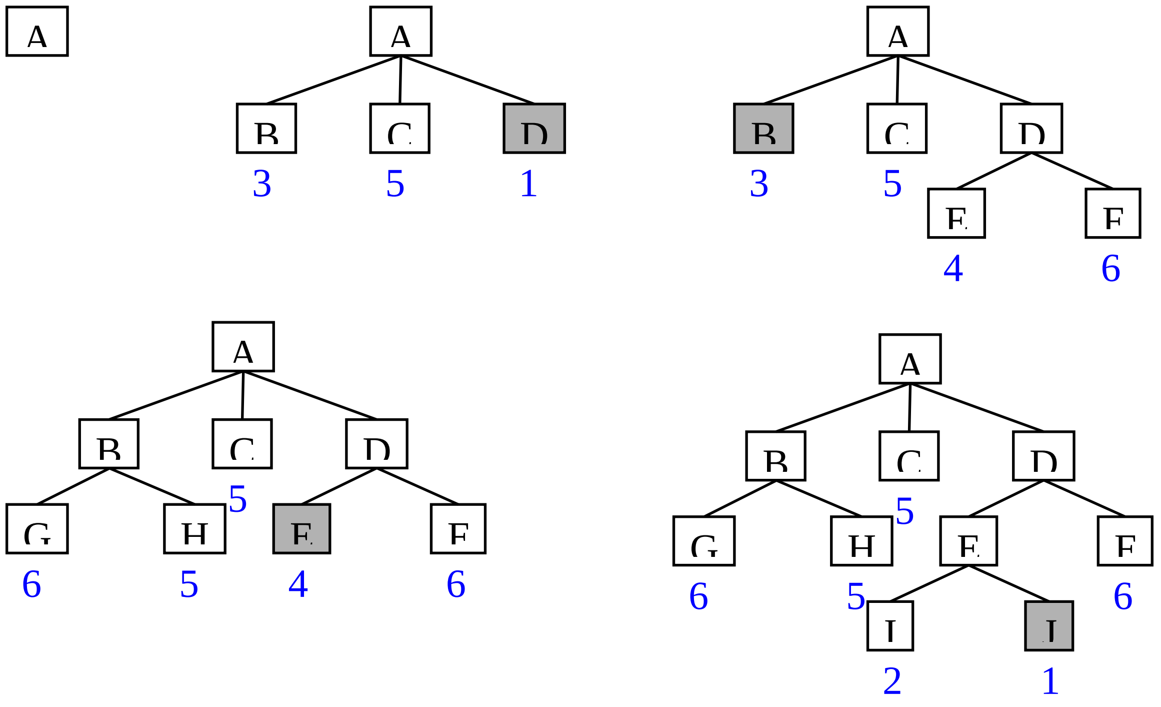
Module 2

Best First Search

Best First Search is a way of combining the advantages of both depth first search and breadth first search into a single method. Depth first search is good because it allows a solution to be found without all competing branches have to be expanded. Breadth first search is good because it does not get trapped on dead end paths. One way of combining two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.



At each step of the best first search process we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution then we can quit. If not all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process is repeated.

Usually what happens is that a bit of depth first searching occurs as the most promising branch is explored. But eventually if a solution is not found, that branch will start to look less promising than one of the top level branches that had been ignored. At that point the now more promising, previously ignored branch will be explored. But the old branch is not forgotten. Its last node remains in the set of generated but unexpanded nodes. The search can return to it whenever all the others get bad enough that it is again the most promising path.

Figure shows the beginning of a best first search procedure. Initially there is only one node, so it will be expanded. Doing so generates 3 new nodes. The heuristic function, which, in this example, is the cost of getting to a solution from a given node, is applied to each of these new nodes. Since node D is the most promising, it is expanded next, producing 2 successor nodes, E and F. But then the heuristic function is applied to them. Now another path, that going through node B, looks more promising, so it is pursued, generating nodes G and H. But again when these new nodes are evaluated they look less promising than another path, so attention is returned to the path through D to E. E is then expanded, yielding nodes I and J. At next step, J will be expanded since it is the most promising. This process can continue until a solution is found.

Although the example above illustrates a best first search of a tree, it is sometimes important to search a graph instead so that duplicate paths will not be pursued. An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space. Each node will contain in addition to a description of the problem state it represents, an indication of how promising it is, a parent link that points back to the best node from which it came, and a list of the nodes that were generated from it.

The parent link will make it possible to recover the path to the goal once the goal is found. The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors. We will call a graph of this sort an OR graph, since each of its branches represents an alternative problem-solving path. To implement such a graph search procedure, we will need to use 2 lists of nodes:

- **OPEN**: nodes that have been generated, and have had the heuristic function applied to them but which have not yet been examined (i.e., had their successors generated). This is actually a priority queue in which the elements with the highest priority are those with the most promising values of the heuristic function.

- **CLOSED**: nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated; we need to check whether it has been generated before.

Algorithm:

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left in OPEN do:
    a) Pick the best node in OPEN
    b) Generate its successors
    c) For each successor do:
        i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent

ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Completeness: Yes. This means that, given unlimited time and memory, the algorithm will always find the goal state if the goal can possibly be found in the graph. Even if the heuristic function is highly inaccurate, the goal state will eventually be added to the open list and will be closed in some finite amount of time.

Time Complexity: It is largely dependent on the accuracy of the heuristic function. An inaccurate h does not guide the algorithm toward the goal quickly, increasing the time required to find the goal. For this reason, in the worst case, the Best-First Search runs in exponential time because it must expand many nodes at each level. This is expressed as $O(b^d)$, where b is the branching factor (i.e., the average number of nodes added to the open list at each level), and d is the maximum depth.

Space Complexity: The memory consumption of the Best-First Algorithm tends to be a bigger restriction than its time complexity. Like many graph-search algorithms, the Best-First Search rapidly increases the number of nodes that are stored in memory as the search moves deeper into the graph. One modification that can improve the memory consumption of the algorithm is to only store a node in the open list once, keeping only the best cost and predecessor. This reduces the number of nodes stored in memory but requires more time to search the open list when nodes are inserted into it. Even after this change, the space complexity of the Best-First Algorithm is exponential. This is stated as $O(b^d)$, where b is the branching factor, and d is the maximum depth.

Optimality: No. The Best-First Search Algorithm is not even guaranteed to find the shortest path from the start node to the goal node when the heuristic function perfectly estimates the remaining cost to reach the goal from each node. Therefore, the solutions found by this algorithm must be considered to be quick estimates of the optimal solutions.

A* Algorithm

A-Star (or A*) is a general search algorithm that is extremely competitive with other search algorithms, and yet intuitively easy to understand and simple to implement. Search algorithms are used in a wide variety of contexts, ranging from A.I. planning problems to English sentence parsing. Because of this, an effective search algorithm allows us to solve a large number of problems with greater ease.

The problems that A-Star is best used for are those that can be represented as a state space. Given a suitable problem, you represent the initial conditions of the problem with an appropriate initial state, and the goal conditions as the goal state. For each action that you can perform, generate successor states to represent the effects of the action. If you keep doing this and at some point one of the generated successor states is the goal state, then the path from the initial state to the goal state is the solution to your problem.

What A-Star does is generate and process the successor states in a certain way. Whenever it is looking for the next state to process, A-Star employs a heuristic function to try to pick the “best” state to process next. If the heuristic function is good, not only will A-Star find a solution quickly, but it can also find the best solution possible.

A search technique that finds minimal cost solutions and is also directed towards goal states is called "**A* (A-star) search**". A* algorithm is a typical heuristic search algorithm, in which the heuristic function is an estimated shortest distance from the initial state to the closest goal state, and it equals to traveled distance plus predicted distance ahead. In Best-First search and Hill Climbing, the estimate of the distance to the goal was used alone as the heuristic value of a state. In A*, we add the estimate of the remaining cost to the actual cost needed to get to the current state.

The A* search technique combines being "guided" by knowledge about where good solutions might be (the knowledge is incorporated into the Estimate function), while at the same time keeping track of how costly the whole solution is.

g(n) = the cost of getting from the initial node to n.
h(n) = the estimate, according to the heuristic function, of the cost of getting from n to the goal node.

That is, **f(n) = g(n) + h(n).** Intuitively, this is the estimate of the best solution that goes through n.

Like breadth-first search, A* is complete in the sense that it will always find a solution if there is one.If the heuristic function h is admissible, meaning that it never overestimates the actual minimal cost of reaching the goal, then A* is itself admissible (or optimal) if we do not use a closed set. If a closed set is used, then h must also be monotonic (or consistent) for A* to be optimal. This means that it never overestimates the cost of getting from a node to its neighbor. Formally, for all paths x,y where y is a successor of x:

$$h(x) \leq g(y) - g(x) + h(y)$$

A* is also **optimally efficient** for any heuristic h, meaning that no algorithm employing the same heuristic will expand fewer nodes than A*, except when there are several partial solutions where h exactly predicts the cost of the optimal path.

A* is both admissible and considers fewer nodes than any other admissible search algorithm, because A* works from an "optimistic" estimate of the cost of a path through every node that it considers -- optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

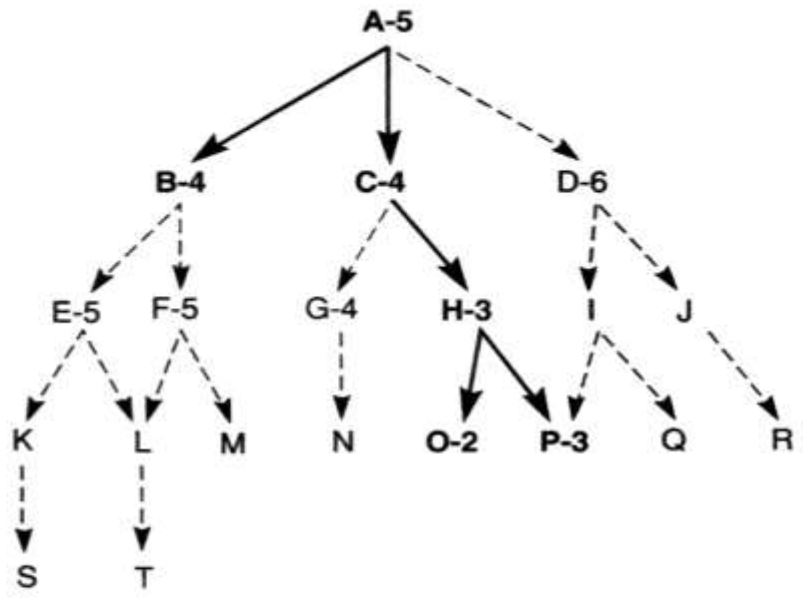
When A* terminates its search, it has, by definition, found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.

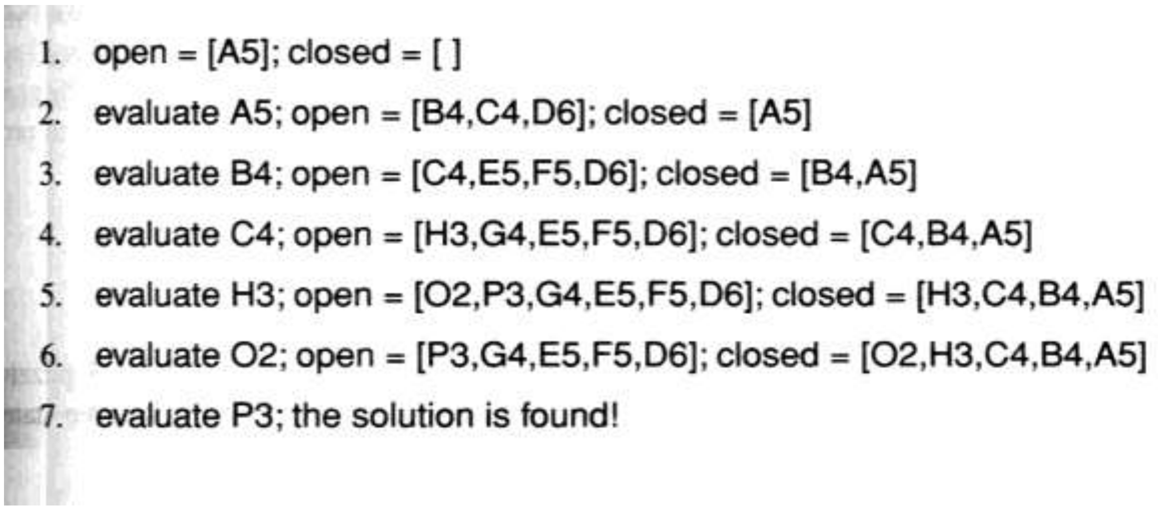
Suppose now that some other search algorithm A terminates its search with a path whose actual cost is not less than the estimated cost of a path through some open node. Algorithm A cannot rule out the possibility, based on the heuristic information it has, that a path through that node might have a lower cost. So while A might consider fewer nodes than A*, it cannot be admissible. Accordingly, A* considers the fewest nodes of any admissible search algorithm that uses a no more accurate heuristic estimate.

Algorithm:

1. Create a search graph, G, consisting solely of the start node N1. Put N1 in a list called OPEN.
2. Create a list called CLOSED that is initially empty.
3. If OPEN is empty, exit with failure.
4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Call this node N.
5. If N is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from N to N1 in G. (The pointers define a search tree and are established in step 7.)
6. Expand node N, generating the set, M, of its successors that are not already ancestors of N in G. Install these members of M as successors of N in G.
7. Establish a pointer to N from each of those members of M that were not already in G (i.e., not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member, Mi, of M that was already on OPEN or CLOSED, redirect its pointer to N if the best path to Mi found so far is through N. For each member of M already on CLOSED, redirect the pointers of each of its descendants in G so that they point backward along the best paths found so far to these descendants.

8. Reorder the list OPEN in order of increasing *f* values. (Ties among minimal *f* values are resolved in favor of the deepest node in the search tree.)
9. Go to step 3.





1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]
3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!

Completeness: Yes, as long as branching factor is finite.

Time Complexity: It is largely dependent on the accuracy of the heuristic function. In the worst case, the A* Search runs in exponential time because it must expand many nodes at each level. This is expressed as $O(b^d)$, where b is the branching factor (i.e., the average number of nodes added to the open list at each level), and d is the maximum depth).

Space Complexity: The memory consumption of the A* Algorithm tends to be a bigger restriction than its time complexity. The space complexity of the A* Algorithm is also exponential since it must maintain and sort complete queue of unexplored options. This is stated as $O(b^d)$, where b is the branching factor, and d is the maximum depth.

Optimality: Yes, if h is admissible. However, a good heuristic can find optimal solutions for many problems in reasonable time.

Advantages:

1. A* benefits from the information contained in the Open and Closed lists to avoid repeating search effort.

2. A*'s Open List maintains the search frontier where as IDA*'s iterative deepening results in the search repeatedly visiting states as it reconstructs the frontier(leaf nodes) of the search.

3. A* maintains the frontier in sorted order, expanding nodes in a best-first manner.

Iterative-Deepening-A* (IDA*)

Iterative deepening depth-first search or **IDDFS** is a state space search strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches d, the depth of the shallowest goal state. Analogous to this iterative deepening can also be used to improve the performance of the A* search algorithm. Since the major practical difficulty with A* is the large amount of memory it requires to maintain the search node lists, iterative deepening can be of considerable service. **IDA*** is a linear-space version of A*, using the same cost function.

Iterative Deepening A* is another combinatorial search algorithm, based on repeated depth-first searches. IDA* tries to find a solution to a given problem by doing a depth-first search up to a certain maximum depth. If the search fails, it is repeated with a higher search depth, until a solution is found. The search depth is initialized to a lower bound of the solution. Like branch-and-bound, IDA* uses pruning to avoid searching useless branches.

Algorithm:

1. Set THRESHOLD = heuristic evaluation of the start state

2. Conduct depth-first search, pruning any branch when its total cost exceeds THRESHOLD. If a solution path is found, then return it.

2. Increment THRESHOLD by the minimum amount it was exceeded and go to step 2.

Like A*, iterative deepening A* is guaranteed to find an optimal solution. Because of its depth first technique IDA* is very efficient with respect to space. IDA* was the first heuristic search algorithms to find optimal solution paths for the 15-puzzle (a 4x4 version of 8-puzzle) within reasonable time and space constraints.

Completeness: Yes, as long as branching factor is finite.

Time Complexity: Exponential in solution length i.e., $O(b^d)$, where b is the branching factor, and d is the maximum depth.

Space Complexity: Linear in the search depth i.e., O(d), where d is the maximum depth.

Optimality: Yes, if h is admissible. However, a good heuristic can find optimal solutions for many problems in reasonable time.

Advantages:
- Works well in domains with unit cost edges and few cycles
- IDA* was the first algorithm to solve random instances of the 15-puzzle optimally
- IDA* is simpler, and often faster than A*, due to less overhead per node
- IDA* uses a left-to-right traversal of the search frontier.

Disadvantages:
- May re-expand nodes excessively in domain with many cycles (e.g., sequence alignment)
- Real valued edge costs may cause the number of iterations to grow very large

3.4 PROBLEM REDUCTION

So far, we have considered search strategies for OR graphs through which we want to find a single, path to a goal. Such structures represent the fact that we will know how to get from a node to a goal state if we can discover how to get from that node to a goal state along any one of the branches leaving it.

3.4.1 AND-OR Graphs

Another kind of structure, the AND-OR graph (or tree), is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution. Just as in an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the briginal problem might be solved. This is why the structure is called not simply an AND graph but rather an AND-OR graph. An example of an AND-OR graph (which also happens to be an AND-OR tree) is given in Fig. 3.6. AND arcs are indicated with a line connecting all the components.

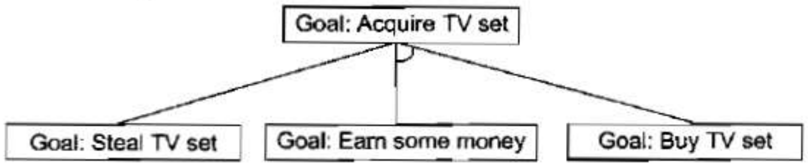


Fig. 3.6 *A Simple AND-OR Graph*

In order to find solutions in an AND-OR graph, we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately. This algorithm should find a path from the starting node of the graph to a set of nodes representing solution states. Notice that it may be necessary to get to more than one solution state since each arm of an AND arc must lead to its own solution node.

To see why our best-first search algorithm is not adequate for searching AND-OR graphs, consider Fig. 3.7(a). The top node, A, has been expanded, producing two arcs, one leading to B and one leading to C and D. The numbers at each node represent the value of f' at that node. We assume, for simplicity, that every operation has a uniform cost, so each arc with a single successor has a cost of 1 and each AND arc with multiple successors has a cost of 1 for each of its components. If we look just at the nodes and choose for expansion the one with the lowest f' value, we must select C. But using the information now available, it would be better to explore the path going through B since to use C we must also use D, for a total cost of 9 (C + D + 2) compared to the cost of 6 that we get by going through B. The problem is that the choice of which node to expand next must depend not only on

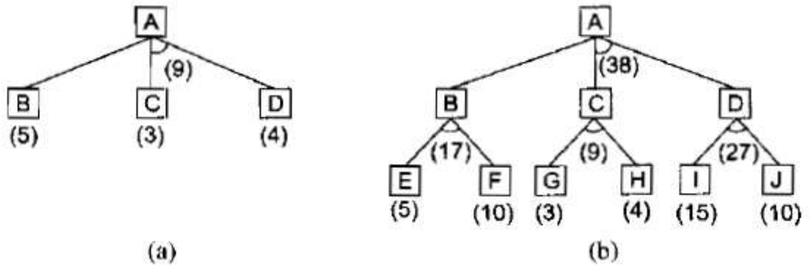


Fig. 3.7 *AND-OR Graphs*

the f' value of that node but also on whether that node is part of the current best path from the initial node. The tree shown in Fig. 3.7(b) makes this even clearer. The most promising single node is G with an f' value of 3. It is even part of the most promising arc G-H, with a total cost of 9. But that arc is not part of the current best path since to use it we must also use the arc I-J, with a cost of 27. The path from A, through B, to E and F is better, with a total cost of 18. So we should not expand G next; rather we should examine either E or F.

In order to describe an algorithm for searching an AND-OR graph, we need to exploit a value that we call *FUTILITY*. If the estimated cost of a solution becomes greater than the value of *FUTILITY*, then we abandon the search. *FUTILITY* should be chosen to correspond to a threshold such that any solution with a cosflibove it is too expensive to be practical, even if it could ever be found. Now we can state the algorithm.

Algorithm: Problem Reduction

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled *SOLVED* or until its cost goes above *FUTILITY*:
   (a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.
   (b) Pick one of these unexpanded nodes and expand it. If there are no successors, assign *FUTILITY* as the value of this node. Otherwise, add its successors to the graph and for each of them compute f' (use only h' and ignore g, for reasons we discuss below). If of any node is 0, mark that node as *SOLVED*.
   (c) Change the f' estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as *SOLVED*. At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. This propagation of revised cost estimates back up the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their f' values be the best estimates available.

This process is illustrated in Fig. 3.8. At step 1, A is the only node, so it is at the end of the current best path. It is expanded, yielding nodes B, C, and D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, which costs 9. (Marked arcs are indicated in the Fig.s by arrows.) In step 2, node D) is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. So we update the f' value of D to 10. Going back one more level, we see that this makes the AND arc B-C better than the arc to D, so it is labeled as the current best path. At step 3, we traverse that arc from A and discover the unexpanded nodes B and C. If we are going to find a solution along this path, we will have to expand both B and C eventually, so let's choose to explore B first. This generates two new arcs, the ones to G and to H. Propagating their f' values backward, we update f' of B to 6 (since that is the best we think we can do, which we can achieve by going through G). This requires updating the cost of the AND arc B-C to 12 (6 + 4 + 2). After doing that, the arc to D is again the better path from A, so we record that as the current best path and either node E or node F will be chosen for expansion at step 4. This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.

In addition to the difference discussed above, there is a second important way in which an algorithm for searching an AND-OR graph must differ from one for searching an OR graph. This difference, too, arises from the fact that individual paths from node to node cannot be considered independently of the paths through other nodes connected to the original ones by AND arcs. In the best-first search algorithm, the desired path

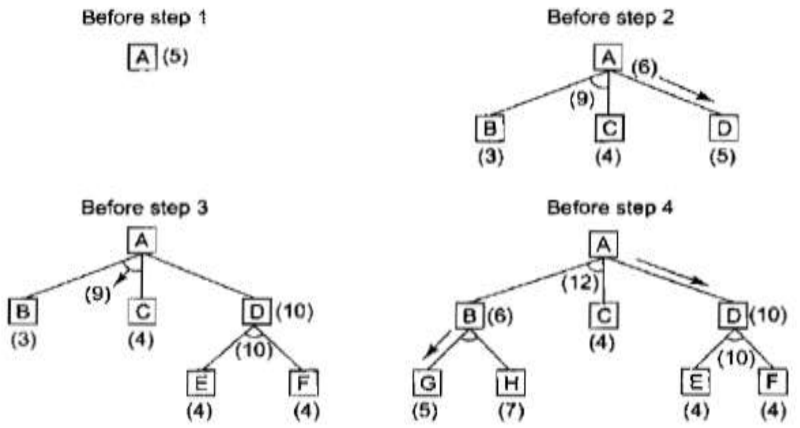


Fig. 3.8 *The Operation of Problem Reduction .*

from one node to another was always the one with the lowest cost. But this is not always the case when searching an AND-OR graph.

Consider the example shown in Fig. 3.9(a). The nodes were generated in alphabetical order. Now suppose that node J is expanded at the next step and that one of its successors is node E, producing the graph shown in Fig. 3.9(b). This new path to E is longer than the previous path to E going through C. But since the path through C will only lead to a solution if there is also a solution to D, which we know there is not, the path through J is better.

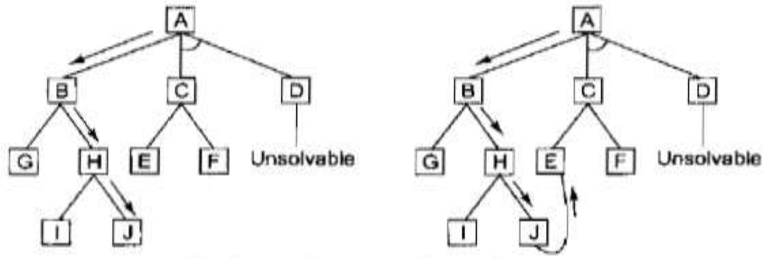


Fig. 3.9 *A Longer Path May Be Better*

There is one important limitation of the algorithm we have just described. It fails to take into account any interaction between subgoals. A simple example of this failure is shown in Fig. 3.10. Assuming that both node C and node E ultimately lead to a solution, our algorithm will report a complete solution that includes both of them. The AND-OR graph states that for A to be solved, both C and D must be solved. But then the algorithm considers the solution of D as a completely separate process from the solution of C. Looking just at the alternatives from D, E is the best path. But it turns out that C is necessary anyway, so it would be better also to use it to satisfy D. But since our algorithm does not consider such interactions, it will find a nonoptimal path. In Chapter 13, problem-solving methods that can consider interactions among subgoals are presented.

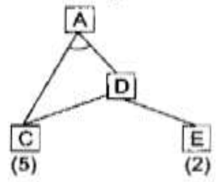


Fig. 3.10 *Interacting Subgoals*

Constraint Satisfaction

Many problems in AI can be viewed as problems of constraint satisfaction in which the goal is to discover some problem states that satisfies a given set of constraints. Examples of this sort of problems include **crypt arithmetic puzzles** and many real world perceptual labeling problems. By viewing a problem as one of constraint satisfaction it is often possible to reduce substantially the amount of search that is required with some other method.

A **Constraint Satisfaction Problem** is characterized by:

- a *set of variables* $\{x_1, x_2, .., x_n\}$,
- for each variable x_i a *domain* D_i with the possible values for that variable, and
- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognizing function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n, a value in D_i for x_i so that all constraints are satisfied.

The **reasons** for choosing to represent and solve a problem as a CSP rather than, say as a mathematical programming problem are twofold.

- Firstly, the representation as a CSP is often much closer to the original problem: the variables of the CSP directly correspond to problem entities, and the constraints can be expressed without having to be translated into linear inequalities. This makes the formulation simpler, the solution easier to understand, and the choice of good heuristics to guide the solution strategy more straightforward.
- Secondly, although CSP algorithms are essentially very simple, they can sometimes find solution more quickly than if integer programming methods are used.

Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained enough. It is a 2 step process. First **constraints are discovered and propagated** as far as possible throughout the system. Then if there is still not a solution, search begins. A **guess about something is made** and added as a new constraint. Propagation can then occur with this new constraint and so forth.

The first step propagation arises from the fact that there are usually dependencies among the constraints. These dependencies occur because many constraints involve more than one object and many objects participate in more than one constraint. For e.g.: assume we start with one constraint N=E+1. Then if we added the constraint N=3 we could propagate that to get a stronger constraint on E namely E=2.

It is fairly easy to see that a CSP can be given an **incremental formulation** as a standard search problem as follows:

Initial state: the empty assignment in which all variables are unassigned.
Successor function: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
Goal test: the current assignment is complete.
Path cost: a constant cost (e.g., 1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables. Furthermore, the search tree extends only to depth n. For these reasons, depth-first search algorithms are popular for CSPs. It is also the case that the path by which a solution is reached is irrelevant. Hence, we can also use a **complete-state formulation**, in which every state is a complete assignment that might or might not satisfy the constraints.

Constraint propagation terminates for one of the 2 reasons. First a **contradiction** may be detected. If this happens then there is no solution consistent with all known constraints. If the contradiction involves only those constraints that were given as a part of problem specification then no solution exists. The second possible reason for termination is that **propagation has run out of stream** and there are no further changes that can be made on the basis of current knowledge.

At this point the second step begins. Some hypothesis about a way to strengthen the constraints must be made. In this case of crypt arithmetic problem this usually means guessing a particular value for some letter. Once this has been done constraint propagation can begin again from this new state. If a solution is found it can be reported. If a contradiction is detected then backtracking can be used to try a different guess and proceed with it.

Algorithm:

1. Propagate available constraints. To do this first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty.
    a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB
    b) If this set is different from the set that was assigned the last time OB was examined or if then first time OB has been examined then add to OPEN all objects that share any constraint with OB.
    c) Remove OB from OPEN.

2. If the union of the constraints discovered above defines a solution the quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction the quit then return failure.
4. If neither of the above occurs then it is necessary to make a guess at something in order to proceed. To do this loop until a solution is found or all possible solutions have been eliminated.
   a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

To apply this algorithm in a particular problem domain it requires the use of 2 kinds of rules. Rules that define the way constraints may validly be propagated and rules that suggest guesses when guesses are necessary. To see how the algorithm works consider the following crypt arithmetic problem

```
  S E N D
  M O R E
=========
M O N E Y
```

We have to replace each letter by a distinct digit so that the resulting sum is correct. Furthermore, numbers must not begin with a zero.

Initial state: No 2 letters have the same value.
The sum of the digits must be as shown in the problem.

The **goal state** is a problem state in which all letters have been assigned a digit in such a way that all the initial constraints are satisfied. The solution proceeds in cycles. At each cycle 2 significant things are done.

1. Constraints are propagated by using rules that correspond to the properties of arithmetic.
2. A value is guessed for some letter whose value is not yet determined.

There are 10!/2 (= 1,814,400) possibilities to check.

<u>Heuristics for Constraint satisfaction</u>

A few useful heuristics can help the best guess to try first. For e.g. if there is a letter that has only 2 possible values and another with 6 possible values, there is a better chance of guessing right on the first than on the second. Another useful heuristic is that if there is a letter that participates in many constraints then it is a good idea to prefer it to a letter that participates in a few. A guess on such a highly constrained letter will usually lead quickly either to a contradiction (if it is wrong) or to the generation of many additional constraints (if it is right).A guess on a less constrained letter on the other hand provides less information.

Games as search problem

- **Initial state:** Initial State is the current board/position
- **Operators:** legal moves and their resulting states
- **A terminal test:** decide if the game has ended
- **A utility function (Payoff function):** produces a numerical value for (only) the terminal states. Example: In chess, outcome = win/loss/draw, with values +1, -1, 0 respectively

Game Trees

The sequence of states formed by possible moves is called a **game tree** ; each level of the tree is called a **ply**. In 2 player games we call the two players **Max** (us) and **Min** (the opponent).WIN refers to winning for Max. At each ply, the "turn" switches to the other player.

Each level of search nodes in the tree corresponds to all the possible board configurations for a particular player – Max or Min. Utility values found at the end can be returned back to their parent nodes. So, winning for Min is *losing* for Max. Max wants to end in a board with +1 and Min in a board with a value of -1.

Max chooses the board with the max utility value, Min the minimum. Max is the first player and Min is the second. Every player needs a **strategy**. For example, the strategy for Max is to reach a winning terminal state regardless of what Min does. Even for simple games, the search tree is huge.

Minimax Algorithm

The **Minimax Game Tree** is used for programming computers to play games in which there are two players taking turns to play moves. Physically, it is just a tree of all possible moves.

With a full minimax tree, the computer could look ahead for each move to determine the best possible move. Of course, as you can see in example diagram shown below, the tree can get very big with only a few moves. Thus, for large games like Chess and Go, computer programs are forced to estimate who is winning or losing by focusing on just the top portion of the entire tree. In addition, programmers have come up with all sorts of **algorithms** and tricks such as Alpha-Beta pruning.

The minimax game tree, of course, cannot be used very well for games in which the computer cannot see the possible moves. So, minimax game trees are best used for games in which both players can see the entire game situation. These kinds of games, such as checkers, othello, chess, and go, are called **games of perfect information**.

For instance, take a look at the following (partial) search tree for Tic-Tac-Toe. Notice that unlike other trees like binary trees, 2-3 trees, and heap trees, a node in the game tree can have any number of children, depending on the game situation. Let us assign points to the outcome of a game of Tic-Tac-Toe. If X wins, the game situation is given the point value of 1. If O wins, the game has a point value of -1. Now, X will be trying to **maximize** the point value, while O will be trying to **minimize** the point value. So, one of the first researchers on the minimax tree decided to name player X as Max and player O as Min. Thus, the entire data structure came to be called the **minimax game tree**.

The games we will consider here are:

- two-person: there are two players.
- perfect information: both players have complete information about the state of the game. (Chess has this property, but poker does not.)
- zero-sum: if we count a win as +1, a tie as 0, and a loss as -1, the sum of scores for both players is always zero.

Examples of such games are chess, checkers, and tic-tac-toe.

This minimax logic can also be extended to games like chess. In these more complicated games, however, the programs can only look at the part of the minimax tree; often, the programs can't even see the end of the game because it is so far down the tree.

So, the computer only looks at a certain number of nodes and then stops. Then the computer tries to **estimate who is winning and losing** in each node, and these estimates result in a numerical point value for that game position. If the computer is playing as Max, the computer will try to maximize the point value of the position, with a win (checkmate) being equal to the largest possible value (positive 1 million, let's say). If the computer is playing as Min, it will obviously try to minimize the point value, with a win being equal to the smallest possible value (negative 1 million, for instance).

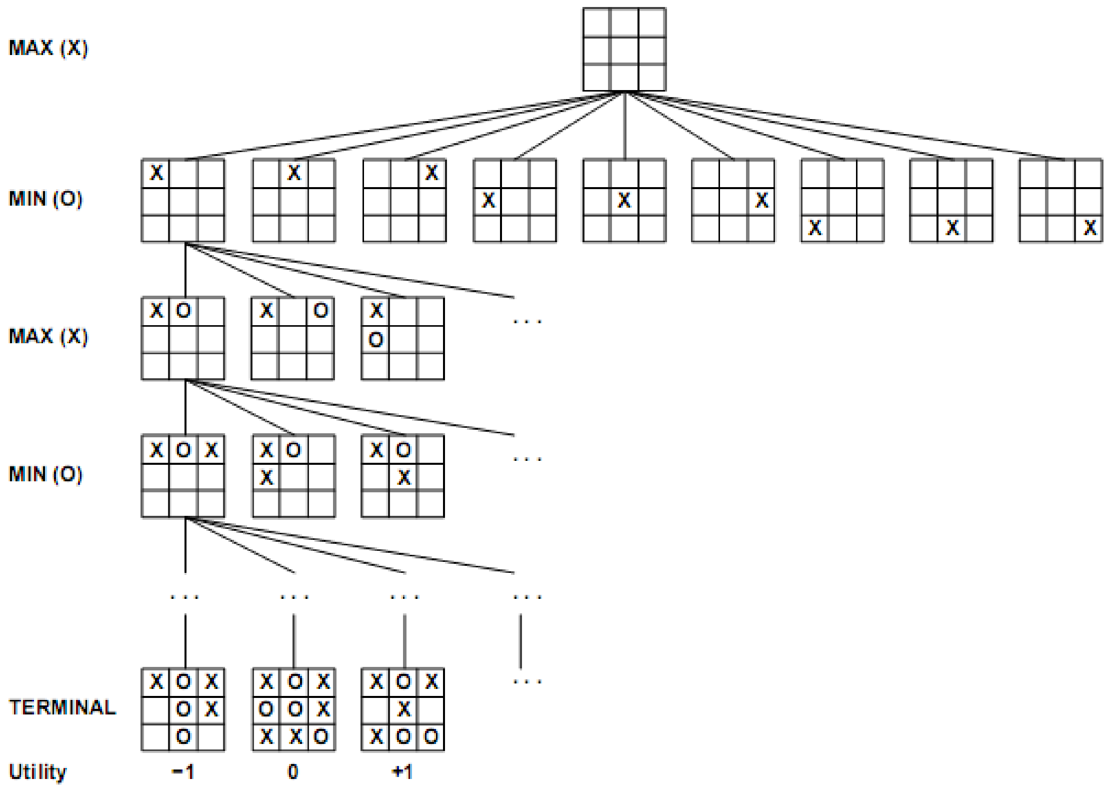


Fig : A (partial)search tree for the game of Tic-Tac-Toe is shown above. The top node is the initial state and max moves first placing an X in an empty square. We show part of the search tree, giving alternating moves by min(O) and max until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Minimax Evaluation

The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, chess, go, and so on. All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premises. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic, they are ‘full information games’. Each player knows everything about the possible moves of the adversary.

- A search tree is generated, depth-first, starting with the current game position up to the end game position.
- Compute the values (through the utility function) for all the terminal states.
- Afterwards, compute the utility of the nodes one level higher up in the search tree (up from the terminal states. The nodes that belong to the MAX player receive the maximum value of its children. The nodes for the MIN player will select the minimum value of its children.
- Continue backing up the values from the leaf nodes towards the root.
- When the root is reached, Max chooses the move that leads to the highest value (optimal move).

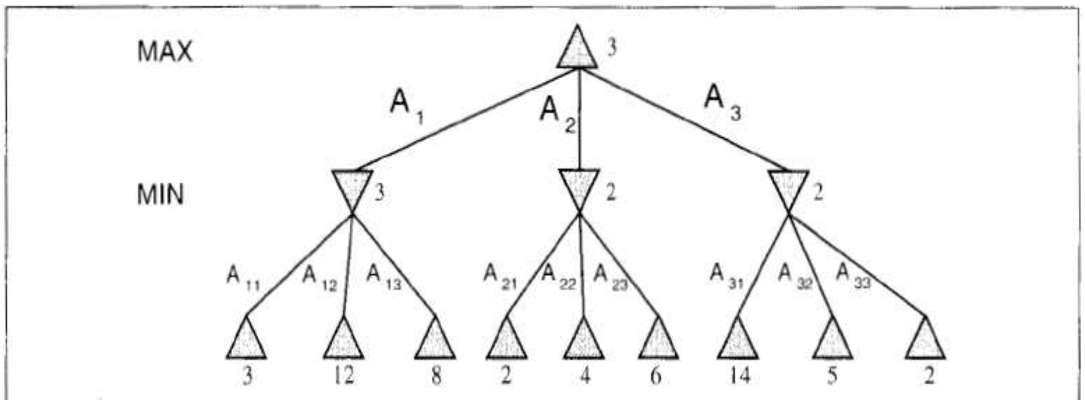
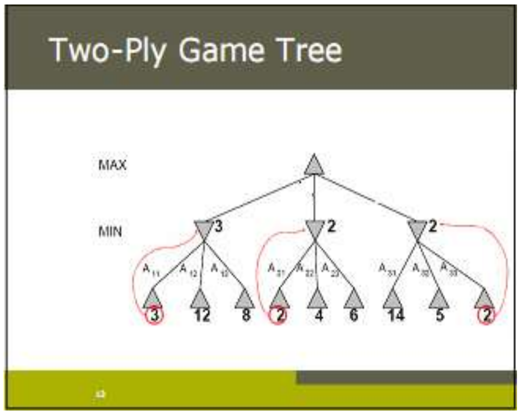
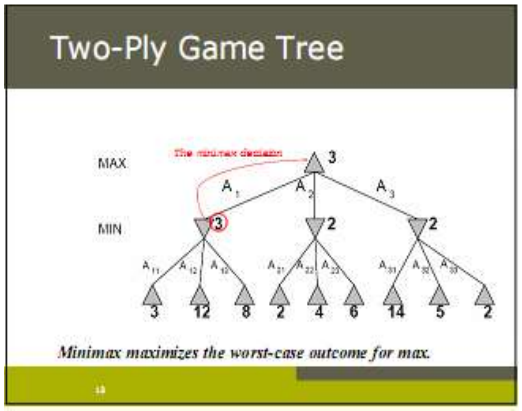


Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX's best move is A_1, and MIN's best reply is A_{11}.

Given a game tree the optimal strategy can be determined by examining the minimax value of each node which we can write as MINIMAX-VALUE (n). Utility value is the value of a terminal node in the game tree. Minimax value of a terminal state is just its utility. Minimax value indicates the *best* value that the current player can possibly get. It's either the max or the min of a **bunch** of utility values. The minimax algorithm is a depth-first search. The space requirements are linear with respect to *b* and *m* where b is the no of legal moves at each point and m is the maximum depth of the tree. For real games, the time cost is impractical.

The minimax search will be as below:





Algorithm:

```
function MINIMAX-DECISION(state) returns an action
    $v \leftarrow$ 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)
    $v \leftarrow -\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s$))
    return $v$
```

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

The MINIMAX value of a node can be calculated as

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

Above shown is an algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The function MAXVALUE and MINVALUE go through the whole game tree all the way to the leaves to determine the backed up value of a state.

Completeness: Yes (if tree is finite)
Time complexity: O(b^d), where d is the depth of the tree.
Space complexity: O(bd) (depth-first exploration), where d is the depth of the tree.
Optimality: Yes, provided perfect info (evaluation function) and opponent is optimal

Alpha-Beta Search

The problem with minimax search is that the no of game states it has to examine is exponential in the no of moves. Minimax helps us look ahead four or five ply in chess. Average human chess players can make plans six or eight ply ahead. But it is possible to compute the correct minimax decision without looking at every node in the game tree. **Alpha-beta pruning** can be used in this context**.** It is similar to the minimax algorithm (applied to the same tree) it returns the same move as minimax would but prunes branches that cannot possibly influence the final decision.

- *alpha* is the value of the best choice (highest value) we have found till now along the path for MAX.
- *beta* is the value of the best choice (lowest value) we have found so far along the path for MIN.

Alpha-beta pruning changes the values for *alpha* or *beta* as it moves and prunes a sub tree as soon as it finds out that it is worse than the current value for *alpha* or *beta.* If two nodes in the hierarchy have incompatible inequalities (no possible overlap), then we know that the node below will not be chosen, and we can stop search.

Implementing Alpha-Beta Search

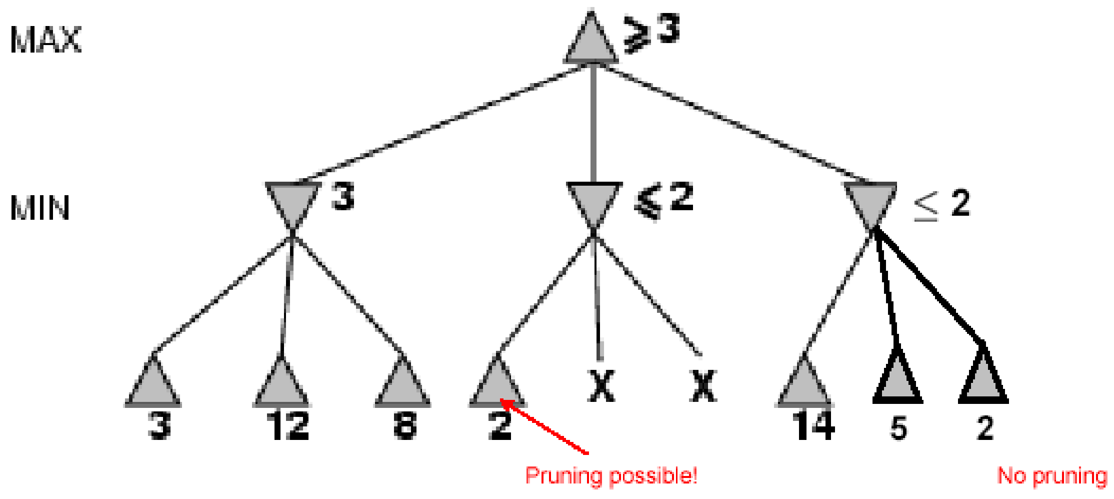
Alpha-beta search is easily implemented by adding *&alpha* and *&beta* parameters to a depth-first minimax search. The alpha-beta search simply quits early and returns the current value when *&alpha* or *&beta* threshold is exceeded. Alpha-beta search performs best if the best moves (for Max) and worst moves (for Min) are considered first; in this case, the search complexity is reduced to $O(b^{d/2})$. For games with high symmetry (e.g. chess), a *transposition table* (Closed list) containing values for previously evaluated positions can greatly improve efficiency.

Alpha-Beta Search Example

Bounded depth-first search is usually used, with the alpha-beta algorithm, for game trees. However:

1. The depth bound may stop search just as things get interesting (e.g. in the middle of a piece exchange in chess. For this reason, the depth bound is usually extended to the end of an exchange.
2. The search may tend to postpone bad news until after the depth bound: the *horizon effect*.

Frequently, large parts of the search space are irrelevant to the final decision and can be pruned. No need to explore options that are already definitely worse than the current best option. Consider again the 2 ply game tree from Fig 5.2. If we go through the calculation of optimal decision once more we can identify the minimax decision without ever evaluating 2 of the leaf nodes.



Let the 2 unevaluated nodes be x and y and let z be the minimum of x and y. The value of root node is given by

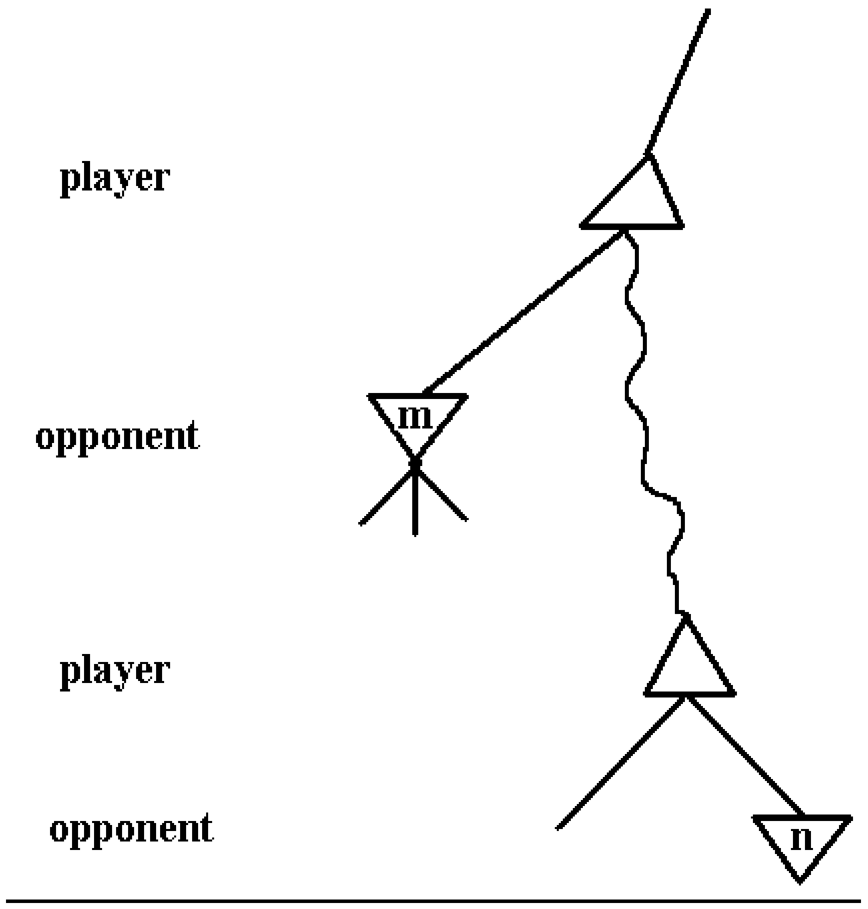
MINIMAX-VALUE (root) = max(min(3,12,8), min(2,x,y),min(14,5,2))

= max(3,min(2,x,y),2)

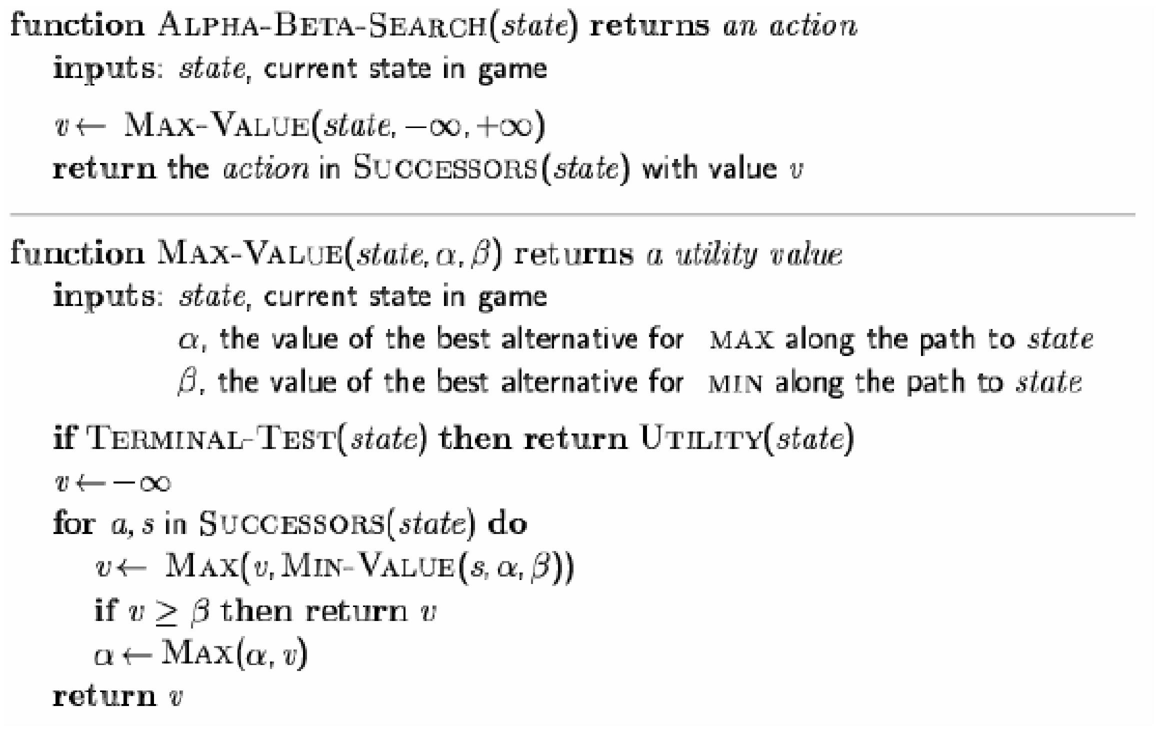
= max(3,z,2) where $z \leq 2$

= 3.

In other words the value of root and hence the minimax decision are independent of the values of the pruned leaves x and y. Alpha beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub trees rather than just leaves. The general principle is this: consider a node n somewhere in the tree such that a player has a choice of moving to that node. If the player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play.

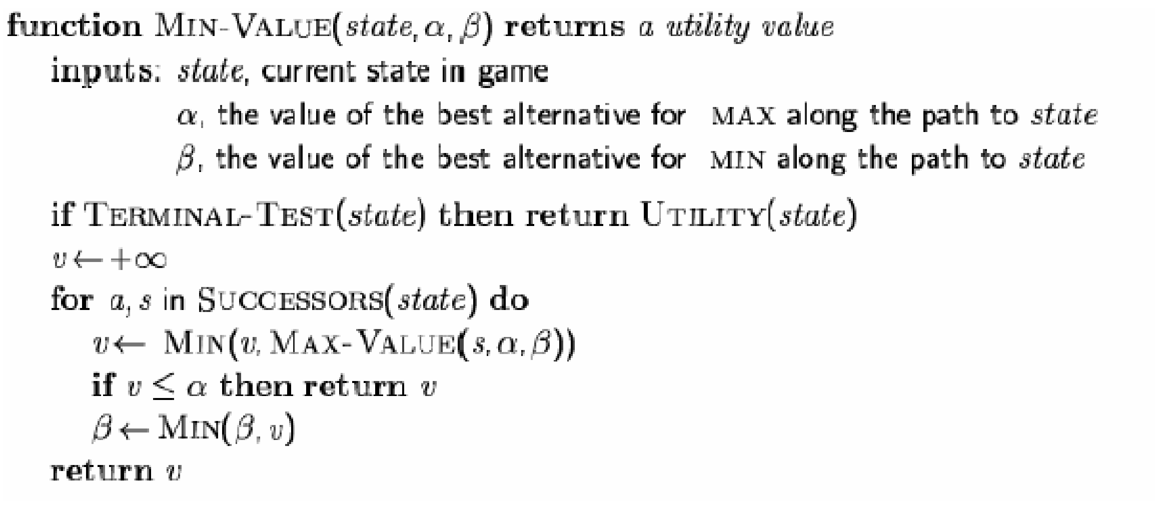


Algorithm:



```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
    return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
    inputs: state, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to state
            $\beta$, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s$, $\alpha$, $\beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha$, $v$)
    return $v$
```



```
function MIN-VALUE($state, \alpha, \beta$) returns a utility value
    inputs: state, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to state
            $\beta$, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow +\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MIN($v$, MAX-VALUE($s, \alpha, \beta$))
        if $v \leq \alpha$ then return $v$
        $\beta \leftarrow$ MIN($\beta, v$)
    return $v$
```

The minimax search is depth first so at any one time we just have to consider the nodes along a single path in the tree. The effectiveness of alpha beta pruning is highly dependent on the order in which the successors are examined. In games repeated states occur frequently because of **transpositions**- different permutations of the move sequence that end up in the same position. It is worthwhile to store the evaluation of this position in a hash table the first time it is encountered so that we don't have to recompute it on subsequent occurrences.

The hash table of previously seen positions is traditionally called transposition table. It is essentially identical to the CLOSED list in graph search. If we are evaluating a million nodes per second, it is not practical to keep all of them in the transposition table. Various strategies can be used to choose the most valuable ones.

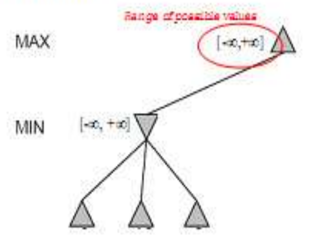
Limitations of alpha-beta Pruning

- Still dependant on search order. Can apply best-first search techniques
- Still has to search to terminal states for parts of the tree
- Depth may not be practical due to time Constraints

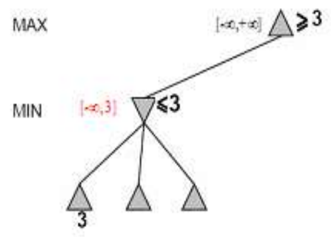
Completeness: Yes, provided the tree is finite
Time complexity: O(b^d) in worst case where d is the depth of the tree. In best case Alpha-Beta, i.e., minimax with alpha-beta pruning, runs in time O($b^{d/2}$) , thus allowing us to evaluate a game tree twice as deep as we could with plain minimax.
Space complexity: O(bd) (depth-first exploration), where d is the depth of the tree
Optimality: yes, provided perfect info (evaluation function) and opponent is optimal
where d: Depth of tree and b: Legal moves at each point

Alpha-Beta Example

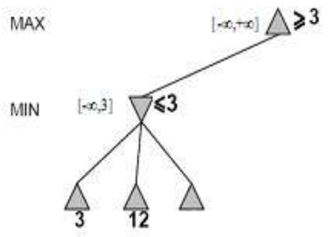
Do DF-search until first leaf



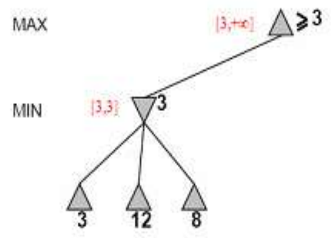
Alpha-Beta Example (continued)

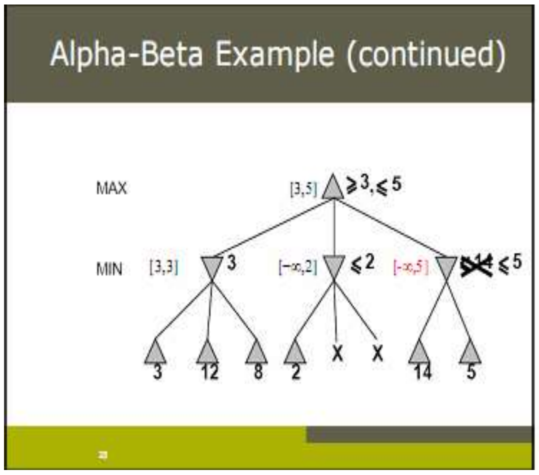


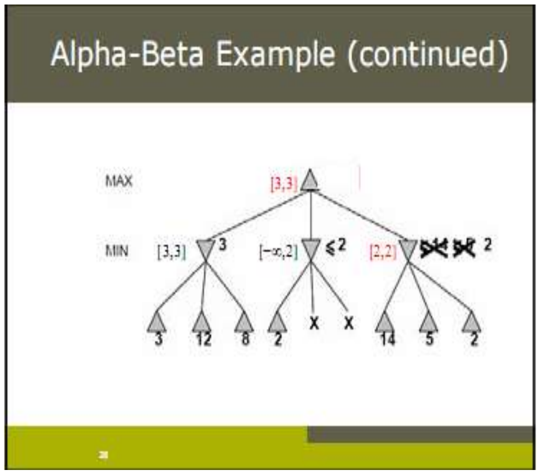
Alpha-Beta Example (continued)

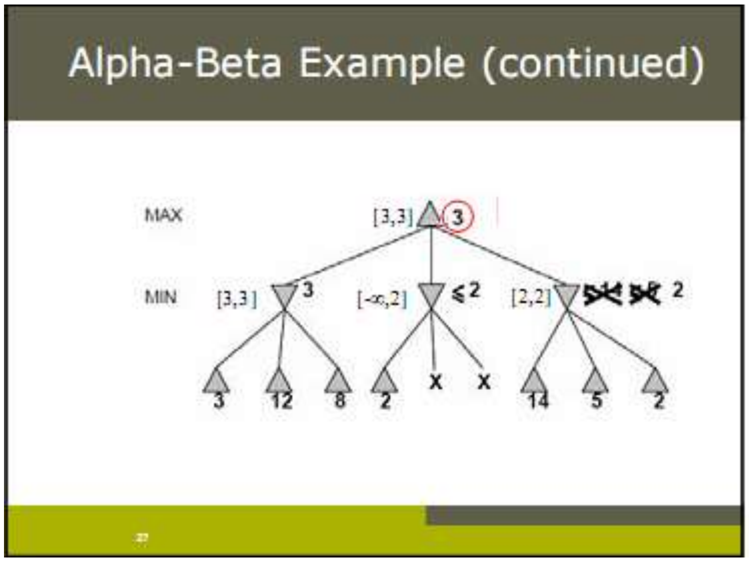


Alpha-Beta Example (continued)



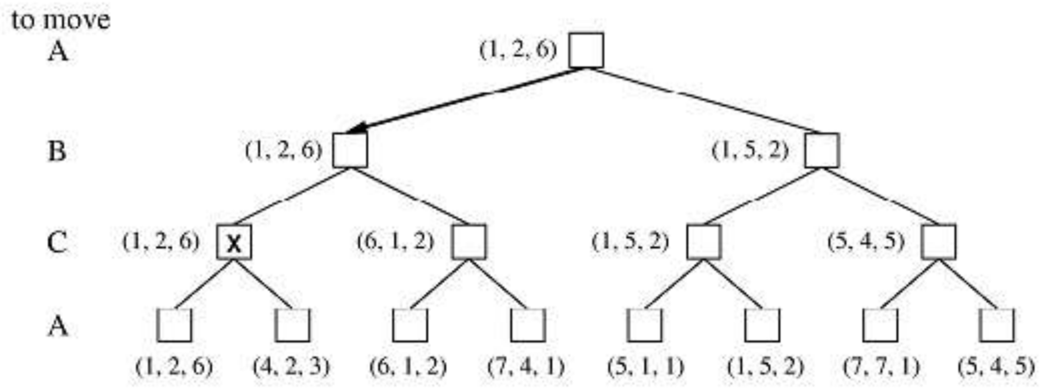

Alpha-Beta Example (continued)


Alpha-Beta Example (continued)


Alpha-Beta Example (continued)

Optimal decisions in multiplayer games

Many popular games allow more than 2 players. Let us examine how to extend minimax idea to multiplayer games. First we need to replace the single values for each node with a vector of values. For e.g. in a 3 player game with players A, B and C a vector $<v_A,v_B,v_C>$ is associated with each node. For terminal state this vector gives the utility of the state from each player's viewpoint. In 2 player games the 2 element vector can be reduced to a single value because the values are always opposite. The simplest way to implement this is to have the utility function return a vector of utilities.



Now we have to consider non terminal states. Consider the node marked X in the game tree shown below. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors <v_A=1,v_B=2,v_C=6> and <v_A=4,v_B=2,v_C=3>. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to terminal states with utilities <v_A=1,v_B=2,v_C=6>. Hence the backed-up value of X is this vector. In general the backed-up value of a node n is the utility vector of whichever successor has the highest value for the player choosing at n.

Anyone who plays multiplayer games quickly becomes aware that there is a lot more going on than in 2 player games. Multiplayer games usually involve alliances, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. In general the backed up value of a node is the utility vector of whichever successor has the highest value for the player choosing at n.

Imperfect Decisions

The minimax algorithm generates the entire game search space whereas the alpha beta algorithm allows us to prune large parts of it. However alpha beta search still has to search all the way to terminal states for at least a portion of the search space. The minimax is impractical since we assume that the program has time to search all the way to the terminal states. Shannon proposed the use of a heuristic evaluation function instead of the utility function that will enable us to stop the search earlier.

Although Alpha-Beta pruning helps to reduce the size of the search space, one still has to reach a terminal node before he can obtain a utility value. In other words, the suggestion is to alter minimax or alpha beta in 2 ways: the utility function is replaced by a heuristic evaluation function EVAL which gives us an estimate of the expected utility of the game from a given viewpoint and the terminal test is replaced by a cutoff test . For example, each pawn in chess is worth 1, a knight or bishop 3, and the queen 9.

Evaluation function

An **evaluation function**, also known as a **heuristic evaluation function** or **static evaluation function**, is a function used by game-playing programs to estimate the value or goodness of a position in the minimax and related algorithms. The evaluation function is typically designed to be fast and accuracy is not a concern (therefore heuristic); the function looks only at the current position and does not explore possible moves (therefore static).

One popular strategy for constructing evaluation functions is as a weighted sum of various factors that are thought to influence the value of a position. For instance, an evaluation function for chess might take the form

$$\mathbf{c_1 * material + c_2 * mobility + c_3 * king\ safety + c_4 * center\ control + ...}$$

Chess beginners, as well as the simplest of chess programs, evaluate the position taking only "material" into account, i.e they assign a numerical score for each piece (with pieces of opposite color having scores of opposite sign) and sum up the score over all the pieces on the board. On the whole, computer evaluation functions of even advanced programs tend to be more materialistic than human evaluations. This, together with their reliance on tactics at the expense of strategy, characterizes the style of play of computers.

Most evaluation functions work by calculating various **features** of a state for e.g. the number of pawns possessed by each side in the game of chess. The features taken together define various categories or equivalence classes of states: the states in each category have the same values for all the features. Any given category will contain some states that lead to wins, some that lead to draws and some that lead to losses. The evaluation function cannot know which states are which but it can return a single value that reflects the proportion of states with each outcome.

The evaluation function should not take long and should comply with the utility function on terminal states. Probabilities can help. For example, a position A with 40% chance of winning, 35% of losing, 25% of being a draw, has as evaluation:

$$\mathbf{1\ x.4+-1x.35+0x.25=.05}$$

In practice this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities of winning. Instead most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value. Mathematically this kind of evaluation function is called a **weighted linear function** is

$$\mathbf{EVAL(s)=w_1f_1(s)+w_2f_2(s)+\dots\dots\dots w_nf_n(s) = \sum_{i=1}^{n} w_if_i(s)}$$

where w_i's are the weights (e.g., 1 for a pawn,5 for a rook,9 for a queen) and f_i 's are the features of a particular position. For chess the f_i could be the number of each kind of piece on the board and the w_i could be the values of the pieces.

Adding up the values of features seems like a reasonable thing to do but in fact it involves a very strong assumption that the contribution of each feature is independent of the other features. A *static evaluation function* evaluates a position without doing any search.

Example 1: Tic-tac-toe

e(p) = nrows(Max) - nrows(Min) where nrows(i) is the number of complete rows, columns, or diagonals that are still open for player i.

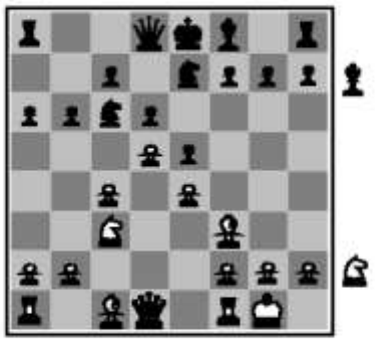
Example 2: Chess

For chess, typically *linear* weighted sum of features

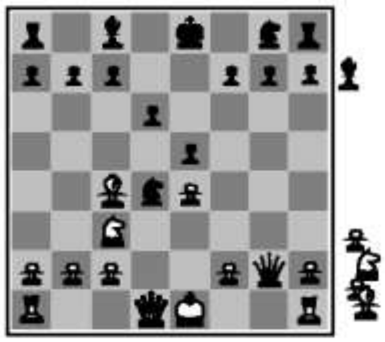
$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) − (number of black queens)
etc.

e(p) = (sum of values of Max's pieces) - (sum of values of Min's pieces) + k * (degree of control of the center)


Black to move

White slightly better


White to move

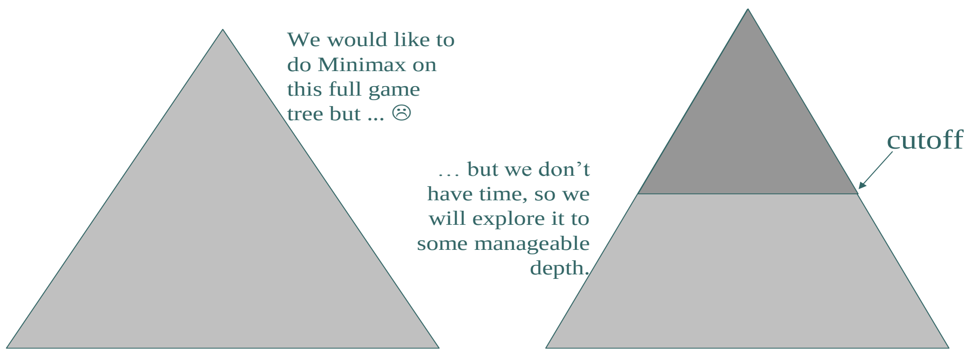
Black winning

Cutting off search

The next step is to modify alpha-beta search so that it will call heuristic EVAL function when it is appropriate to cut off the search.In terms of implementation we replace the 2 lines that mention TERMINAL-TEST with the following line.

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

E.g., Depth limited Minimax search.


We would like to do Minimax on this full game tree but ... ☹
… but we don't have time, so we will explore it to some manageable depth.
cutoff

We must also arrange some bookkeeping so that the current depth is incremented on each recursive call. The most straight forward approach to controlling the amount of search is to set a fixed depth limit, so that CUTOFF-TEST returns true for all depth greater than some fixed depth d. Linear evaluation functions are used often but nonlinear ones are not uncommon. One should tune the weights by “trial-and-error”. Another way is to apply iterative deepening. Heuristic functions evaluate board without knowing where *exactly* it will lead to. It is used to estimate the *probability* of winning from that node.

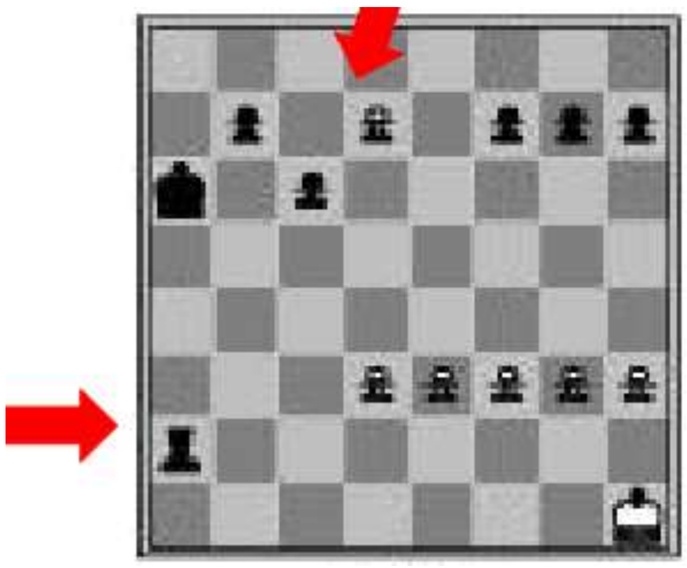
When searching a large game tree (for instance using minimax or alpha-beta pruning) it is often not feasible to search the entire tree, so the tree is only searched down to a certain depth. This results in the horizon effect where a significant change exists just over the "horizon" (slightly beyond the depth the tree has been searched). Evaluating the partial tree thus gives a misleading result. The **horizon effect** is more difficult to eliminate. It arises when the program is facing a move by opponent that causes serious damage and is ultimately unavoidable.

An example of the horizon effect occurs when some negative event is inevitable but postponable, but because only a partial game tree has been analyzed, it will appear to the system that the event can be avoided when in fact this is not the case. For example, in chess, if the white player is one move away from queening a pawn, the black AI can play moves to stall white and push the queening "over the horizon", and mistakenly believe that it avoided it entirely.

The horizon effect can be avoided by using "singular extension" in addition to the search algorithm. A singular extension is a move that is clearly better than all other moves in a given position. This gives the search algorithm the ability to look beyond its horizon, but only at moves of major importance (such as the queening in the previous example). This does not make the search much more expensive, as only a few specific moves are considered. In chess, the computer player may be looking ahead 20 moves. If there are subtle flaws in its position that only matter after 40 moves, then the computer player can be beaten.

Horizon Effect

Fixed depth search thinks it can avoid the queening move



Black to move

The evaluation function should be applied only to positions which are **quiescent** (no dramatic changes in the near future). **Quiescence search** deals with the expansion of nonquiescent positions in order to reach quiescent positions. For example, one may try to postpone the queening move of a pawn “over the horizon” (where it cannot be detected). Essentially, a quiescent search is an evaluation function that takes into account some dynamic possibilities. Sometimes it is restricted to consider only certain types of moves such as capture moves that will quickly resolve the uncertainties in the position.

So far we have talked about cutting off search at a certain level and about doing alpha beta pruning that provably has no effect on the result. It is also possible to do forward pruning meaning that some moves at a given node are pruned immediately without further consideration. Clearly most humans playing chess only consider a few moves from each position. Unfortunately this approach is rather dangerous because there is no guarantee that the best move will not be pruned away. This can be disastrous if applied near the root, because every so often the program will miss some obvious moves. Combining all the techniques described here results in a program that can play creditable chess.