Algorithm: Generate and Test

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.
2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit. Otherwise, return to step I.

Algorithm: Simple Hill Climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until there are no new operators left to be applied in the current slate:
(a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
(b) Evaluate the new state.
(i) If it is a goal state, then return it and quit.
(ii) If it is not a goal state but it is better than the current state, then make it the current state.
(iii) If it is not better than the current state, then continue in the loop.

Algorithm: Steepest – Ascent Hill Climbing (or) Gradient Search

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
(a) Let SUCC be a suite such that any possible successor of the current state will be better than SUCC.
(b) For each operator that applies to the current state do:
(i) Apply the operator and generate a new state.
(ii) Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
(c) If the SUCC is better than current state, then set current state to SUCC.

Algorithm: Simulated Annealing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialize best-so-far to the current state:
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.
    (a)Select an operator that has not yet been applied to the current state and apply it to produce a new state.
    (b) Evaluate the new state. Compute:
    E= (value of current) – (value of new state).

  - If the new state is a goal state, then return it and quit.
  - If it is not a goal state but it is better than the current state, then make it the current state and set best-so-far to this new state.
  - If it is not better than the current state then make it the current state with probability p’ by invoking randomness in the range[0, 1]. If this value is less than p’ then the move is accepted. Otherwise do nothing.

(c) Revise T as necessary according to the annealing schedule.

5. Return best-so-far as the answer.

OR Graphs: Introduction

Depth-first search is good because it finds a solution without expanding all competing branches. Breadth-first search is good because it does not get trapped on dead-end paths. In best first search, the advantages of these two methods are combined together to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current path. At each step of the best-first search process, the most promising of the nodes we have generated so far is selected. This is done by applying an appropriate heuristic function to each state and the implementation is carried out using graph search (to avoid repeated states) with two lists of nodes in queue representation.

- OPEN - nodes that have been generated using the heuristic function and not yet examined (i.e.. had their successors generated). OPEN is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function. Standard techniques for manipulating priority queues can be used to manipulate the list.
- 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 or not.

Algorithm: Best First Search

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

The A* Algorithm

1. Start with *OPEN* holding the initial node with f’ value (h’ + 0). Set *CLOSED* to empty list.
2. Pick the *BEST* node on *OPEN* such that $f = g + h'$ is minimal.
3. If *BEST* node is goal node quit and return the path from initial to *BEST* **Otherwise**
4. Generate the *SUCCESSOR* nodes from *BEST* node and do the following conditions.
   - Update the cost of each *SUCCESSOR* node
   - Check, the *SUCCESSOR* node is already exists in *OPEN* or not. If exists, select the node with minimum cost from initial state and update *OPEN* list.

- If no *SUCCESSOR* nodes are in *OPEN* or *CLOSED* then add it to *OPEN* list as *BEST* node's successor.

Problem Reduction Search: Introduction

Problem reduction search is a basic problem-solving technique of AI. It involves reducing a problem to a set of easier subproblems whose solutions, if found, can be combined to form a solution to the hard problem. Such a search is easily written as a recursive program in Lisp:

- If the given problem is a *primitive subproblem* (one that can be solved by a known technique), return the solution to it.
- Otherwise, try breaking the given problem into sets of simpler *subproblems,* and call the program recursively to try to solve the subproblems .
- If a set of subproblems is found such that all the subproblems can be solved, *combine* the subproblem solutions in an appropriate way to form the solution to the current problem.

The search algorithm of AND-OR graph uses the term *FUTILITY,* that is any solution above the threshold value of *FUTILITY* is too expensive even the solution can be found.

Algorithm : Problem Reduction

1. Initialize the graph with the start node.
2. Loop until the starting node is labeled *SOLVED* or until its cost goes above *FUTILITY*:
(a) Traverse the graph, from the initial node along the current best path that are not 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 comput f' to each of them. If f' of any node is 0, mark that node as *SOLVED.*
(c) Compute the f' value of the newly expanded node and propagate this change backward through the graph.
    - If any nodes descendants are all solved, label the node itself as SOLVED.
    - At each node that is visited update the value depends on most promising and current best path.

The expanded nodes must be re-examined so that the best current path can be selected. Thus it is important that every node should have their best estimate value.

AO* Algorithm

1. Initialise the graph to start node
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
3. Pick any of these nodes and expand it and if it has no successors call this value *FUTILITY* otherwise calculate only *f'* for each of the successors.
4. If *f'* is 0 then mark the node as *SOLVED*
5. Change the value of *f'* for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as *SOLVED* then mark the parent node as *SOLVED*.
7. If starting node is *SOLVED* or value greater than *FUTILITY*, stop, else repeat from 2.

Algorithm : Constraint Satisfaction

1. Propagate available constraints. Assign values to set of all objects and keep it in *OPEN*. Then do the following steps until an inconsistency is detected or until OPEN is empty:
(a) Select an object *OBJ* from OPEN. Strengthen as much as possible the set of constraints that apply to *OBJ*.
(b) If this set is different from the set that was assigned the last time *OBJ* was examined or if this is the first time *OBJ* has been examined, then add to OPEN all objects that share any constraints with *OBJ*.
(c) Remove *OBJ* from OPEN.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, 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

Algorithm: Means-Ends-Analysis (*CURRENT, GOAL*)
I. Compare *CURRENT* to *GOAL*. If there are no differences between them then return.
2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is returned:
(a) Select a suitable operator *O* that is applicable to the current difference. If there are no such operators, then signal failure.
(b) Attempt to apply *O* to *CURRENT*. Generate descriptions of two states: *O-START*, a state in which *O*'s preconditions are satisfied and *O-RESULT*, the state that would result if *O* were applied in *O-START*.
(c) If (*FIRST-PART* <- MEA(*CURRENT, O-START*))
and (*LAST-PART* <- MEA(*O-RESULT, GOAL*))
are successful, then signal success and return the result of concatenating *FIRST-PART*, *O* and *LAST-PART*.