3 Heuristic Search Techniques

Syllabus

Generate-And-Test, Hill Climbing, Best-First Search, Problem Reduction, Constraint Satisfaction, Means-Ends Analysis, A and AO* search.*

Contents

3.1 The Searching Techniques

GTU : Summer-12,14,15,16,18,19,20, Winter-12,14,15,16,17,18,19

3.1.1 Introduction to Informed Search

- In the previous chapter 2 we have seen the concept of uninformed or blind search strategy. Now we will see more efficient search strategy, the informed search strategy. Informed search strategy is the search strategy that uses problem-specific knowledge beyond the definition of the problem itself.
- The general, method followed in informed search is best first search. Best first search is similar to graph search or tree search algorithm wherein node expansion is done based on certain criteria.

3.1.2 Generate-and-Test

- Generate-and-Test is a search algorithm which uses depth first search technique. It assures to find solution in systematic way. It generates complete solution and then the testing is done. A heuristic is needed so that the search is improved.

Following is the algorithm for Generate-and-Test :

1) Generate a possible solution which can either be a point in the problem space or a path from the initial state.
2) Test to see if this possible solution is a real (actual) solution by comparing the state reached with the set of goal states.
3) If it is real solution then return the solution otherwise repeat from state 1.

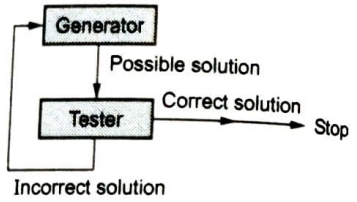


Fig. 3.1.1 Generate-And-Test

Following diagram illustrates the algorithm steps :

- Generate-and-Test is acceptable for simple problems whereas it is inefficient for problems with large spaces.

3.1.3 Best First Search Technique (BFS)

1) Search will start at root node.
2) The node to be expanded next is selected on the basis of an evaluation function, f(n).
3) The node having lowest value for f(n) is selected first. This lowest value of f(n) indicates that goal is nearest from this node (that is f(n) indicates distance from current node to goal node).

Implementation : BFS can be implemented using priority queue where fringe will be stored. The nodes in fringe will be stored in priority queue with increasing value of f(n) i.e. ascending order of f(n). The high priority will be given to node which has low f(n) value.

As name says "best first" then, we would always expect to have optimal solution. But in general BFS indicates that choose the node that appears to be best according to the evaluation function. **Hence the optimality based on 'best-ness' of evaluation function.**

Algorithm for best first search

1. Use two ordered lists OPEN and CLOSED.
2. Start with the initial node 'n_1' and put it on the ordered list OPEN.
3. Create a list CLOSED. This is initially an empty list.
4. If OPEN is empty then exit with failure.
5. Select first node on OPEN. Remove it from OPEN and put it on CLOSED. Call this node n.
6. If 'n' is the goal node exit. The solution is obtained by tracing a path backward along the arcs in the tree from 'n' to 'n_1'.
7. Expand node 'n'. This will generate successors. Let the set of successors generated, be S. Create arcs from 'n' to each member of S.
8. Reorder the list OPEN, according to the heuristic and go back to step 4.

Consider the following 8-puzzle problem -

Here the heuristic used could be "number of tiles not in correct position" (i.e. number of tiles misplaced). In this solution the convention used is, that smaller value of the heuristic function 'f' leads earlier to the goal state.

Step 1

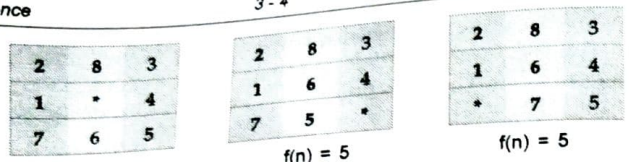
2	8	3
1	6	4
7	*	5

f(n) = 4

This is the initial state where four tiles (1, 2, 6, 8) are misplaced so value of heuristic function at this node is 4.

Step 2

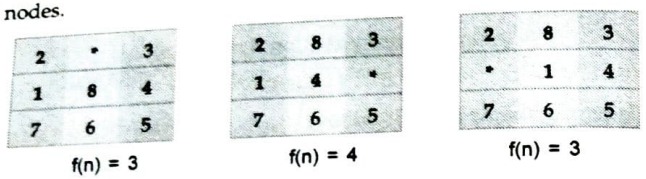
This will be the next step of the tree.



Heuristic function gives the values as 3, 5 and 5 respectively.

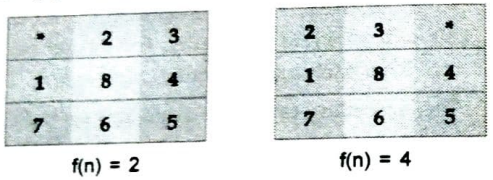
Step 3

Next, the node with the lowest f(n) value 3 is the one to be further expanded which generates 3 nodes.

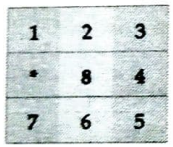


Step 4

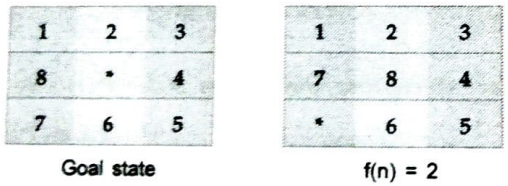
Here there is tie for f(n) value. We continue to expand node.



Step 5



Step 6



In this algorithm a depth factor g is also added to h.

3.1.4 Heuristic Function

- There are many different algorithms which employ the concept of best first search. The main difference between all the algorithms is that they have different evaluation function. The central component of these algorithms is heuristic function - h(n) which is defined as, h(n) = estimate cost of the cheapest path from node 'n' to a goal node.

Note

1) Heuristic function is key component of best first search. It is denoted by h(n) and h(n) = The shortest and cheapest path from initial node to goal node.
2) One can give additional knowledge about the problem to the heuristic function.

   For example - In our problem of Pune-Chennai route we can give information about distances between cities to the heuristic function.
3) A heuristic function guide the search in an efficient way.
4) A heuristic function h(n) consider a node as input but it rely only on the state at that node.
5) h(n) = 0, if n is goal state.

3.1.5 Greedy Best First Search (GBFS)

- Greedy best first search expand the node that is closest to the goal, expecting to get solution quickly :
- It evaluates node by using the heuristic function f(n) = h(n).
- It is termed as greedy (asking for more) because at each step it tries to get as close to the goal as it can.
- GBFS resembles DFS in the way that it prefers to follow a single path all the way to the goal. It back tracks when it comes to dead end that is, to a node from which goal state cannot be reached.
- Choosing minimum h(n) can lead to bad start as it may not yield always a solution. Also as this exploration is not leading to solution, therefore unwanted nodes are getting expanded.
- In GBFS, if repeated states are not detected then the solution will never found.

Performance measurement

1) Completeness : It is incomplete as it can start down an infinite path and never return to try other possibilities which can give solution.

2) Optimal : It is not optimal as it can initially select low value h(n) node but it may happen that some greater value node in current fringe can lead to better solution. Greedy strategies, in general, suffers from this, "looking for current best they loose on future best and inturn finally the best solution" !

3) Time and space complexity : Worst case time and space complexity is $O(b^m)$ where 'm' is the maximum depth of the search space.

The complexity can be reduced by devicing good heuristic function.

3.1.6 A* Search

The A* algorithm

1) A* is most popular form of best first search.
2) A* evaluates node based on two functions namely
   1) g(n) - The cost to reach the node 'n'.
   2) h(n) - The cost to reach the goal node from node 'n'.

These two function's costs are combined into one, to evaluate a node. New function f(n) is deviced as,

$f(n) = g(n) + h(n)$ that is,

f(n) = Estimated cost of the cheapest solution through 'n'.

Working of A*

1) The algorithm maintains two sets.
   a) OPEN list : The OPEN list keeps track of those nodes that need to be examined.
   b) CLOSED list : The CLOSED list keeps track of nodes that have already been examined.
2) Initially, the OPEN list contains just the initial node, and the CLOSED list is empty. Each node n maintains the following : g(n), h(n), f(n) as described above.
3) Each node also maintains a pointer to its parent, so that later the best solution, if found, can be retrieved. A* has a main loop that repeatedly gets the node, call it 'n', with the lowest f(n) value from the OPEN list. If 'n' is the goal node, then we are done, and the solution is given by backtracking from 'n'. Otherwise, 'n' is removed from the OPEN list and added to the CLOSED list. Next all the possible successor nodes of 'n' are generated.
4) For each successor node 'n', if it is already in the CLOSED list and the copy there has an equal or lower 'f' estimate, and then we can safely discard the newly generated 'n' and move on. Similarly, if 'n' is already in the OPEN list and the

copy there has an equal or lower f estimate, we can discard the newly generated n and move on.

5) If no better version of 'n' exists on either the CLOSED or OPEN lists, we remove the inferior copies from the two lists and set 'n' as the parent of 'n'. We also have to calculate the cost estimates for n as follows :

   Set g(n) which is g(n) plus the cost of getting from n to n;

   Set h(n) is the heuristic estimate of getting from n to the goal node ;

   Set f(n) is g(n) + h(n)

6) Lastly, add 'n' to the OPEN list and return to the beginning of the main loop.

Performance measurement for A*

1) Completeness : A* is complete and guarantees solution.

2) Optimality : A* is **optimal if h(n)** is admissible heuristic. **Admissible heuristic** means h(n) never over estimates the cost to reach the goal. Tree-search algorithm gives optimal solution if h(n) is admissible.

If we put extra requirement on h(n) which is consistency also called as monotonicity then we are sure expect the optimal solution. A heuristic function h(n) is said to be consistent, if, for every node 'n' and every successor 'ns' of 'n' generated by action 'a', the estimated cost of reaching the goal from 'n' is not greater than the step cost of getting to 'ns'.

$h(n) \leq \text{cost}(n, a, ns) + h(ns)$

A* using graph-search is optimal if h(n) is consistent.

Note : The sequence of nodes expanded by A* using graph-search is in, increasing order of f(n). Therefore the first goal node selected for expansion must be an optimal solution because all latter nodes will be either having same value of f(n) (same expense) or greater value of f(n) (more expensive) than currently expanded node.

A* never expands nodes with $f(n) > C^*$ (where C^* is the cost of optimal solution).

A* algorithm also do pruning of certain nodes.

Pruning means ignoring a node completely without any examination of that node, and thereby ignoring all the possibilities arising from these node.

3) Time and space complexity : If number of nodes reaching to goal node grows exponentially then time taken by A* eventually increases.

The main problem area of A* is memory, because A* keeps all generated nodes in memory.

A* usually runs out of space long before it runs out of time.

A* optimality proof

- Assume A* finds the (sub optimal) goal G2 and the optimal goal is G.
- Since h is admissible

  $h'(G2) = h'(G) = 0$
- Since G2 is not optimal

  $f(G2) > f(G)$
- At some point during the search, some node 'n' on the optimal path to G is not expanded.

We know,

$$f(n) \leq f(G)$$

3.1.7 Memory Bounded Heuristic Search

If we use idea of Iterative deepening search then we can reduce memory requirements of A*. From this concept we device new algorithm.

Iterative deepening A*

- Like IDDFS uses depth as cutoff value in IDA* f-cost (g+h) is used as cutoff rather than the depth.
- It reduces memory requirements, incurred in A*, thereby putting bound on memory hence it is called as memory bounded algorithm.
- IDA* suffers from real value costs of the problem.
- We will discuss two memory bounded algorithm -
  1) Recursive breadth first search.
  2) MA* (Memory bounded A*).

3.1.7.1 Recursive Best First Search (RBFS)

- It works like best first search but using only linear space.
- Its structure is similar to recursive DFS but instead of continuing indefinitely down the current path it keeps track of the F value of the best alternative path available from any ancestor of the current node.
- The recursion procedure is unwinded back to the alternative path if the current node crosses limit.
- The important property of RBFS is that it remembers the f-value of the best leaf in the forgotten subtree (previously left unexpanded). That is, it is able to take decision regarding re-expanding the subtree.

- It is reliable, cost effective than IDA but its critical problem is excessive node generation.

Performance measure

1) **Completeness :** It is complete.
2) **Optimality :** Recursive best-first search is optimal if h(n) is admissible.
3) **Time and space complexity :** Time complexity of RBFS depends on two factors -
   a) Accuracy of heuristic function.
   b) How often (frequently) the best path changes as nodes are expanded.

RBFS suffers from problem of expanding repeated states, as the algorithm fails to detect them.

Its space complexity is O(bd).

Its suffers from problem of using too little (very less) memory. Between iterations, RBFS maintains more information in memory but it uses only **O(bd) memory**. If more memory is available then also RBFS cannot utilize it.

3.1.7.2 MA*

RBFS under utilizes memory. To overcome this problem MA* is deviced.

Its more simplified version called as simplied MA* proceeds as follows -

1) **If expands the best leaf until memory is full.**
2) At this point it cannot add new node to the search tree without dropping an old one.
3) **It always drops the node with highest f-value.**
4) If goal is not reached then it backtracks and go to the alternative path. In this way the ancestor of a forgotten subtree knows the quality of the best path in that subtree.
5) While selecting the node for expansion it may happen that two nodes are with same f-value. Some problem arises, when the node is discarded (multiple choices can be there as many leaves can have same f-value).

The SMA* generates new best node and new worst node for expansion and deletion respectively.

Performance measurement

1) **Completeness :** SMA* is complete and guarantee solution.
2) **Optimality :** If solution is reachable through optimal path it gives optimal solution. Otherwise it returns best reachable solution.

3) **Time and space complexity :** There is memory limitations on SMA*. Therefore it has to switch back and forth continually between a set of candidate solution paths, only small subset of which can fit in memory. This problem is called as thrashing because of which algorithm takes more time.

Extra time is required for repeated regeneration of the same nodes, means that the problem that would be practically solvable by A* (with unlimited memory) became intractable for SMA*.

It means that memory restrictions can make a problem intractable from the point of view of computation time.

3.1.8 AO* Search and Problem Reduction Using AO*

When a problem can be divided in a set of sub problems, where each sub problem can be solved seperately and a combination of these will be a solution. AND-OR graphs or AND-OR trees are used for representing the solution.

AND-OR graphs

1) The decomposition of the problem or problem reduction generates AND arcs.
2) One AND arc may point to any number of successor nodes.
3) All these must be solved so that the arc will give rise to many arcs, indicating several possible solutions. Hence the graph is known as AND-OR instead of AND.

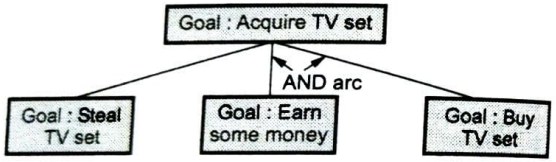


Fig. 3.1.2 [AND-OR graph example]

4) AO* is a best-first algorithm for solving problems represented as a cyclic AND/OR graphs problems.
5) An algorithm to find a solution in an AND-OR graph must handle AND area appropriately.

The comparative study of A* and AO*

1) Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G.
2) G represents the part of the search graph generated so far.
3) Each node in G points down to its immediate successors and upto its immediate predecessors, and also has with it the value of 'h' cost of a path from itself to a set of solution nodes.

4) The cost of getting from the start nodes to the current node 'g' is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state.
5) AO* algorithm serves as the estimate of goodness of a node.
6) A* algorithm cannot search AND-OR graphs efficiently.
7) AO* will always find minimum cost solution.

- The algorithm for performing a heuristic search of an AND-OR graph is given below.

AO* algorithm

1) Initialize 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 as, 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.

3.1.9 How to Search Better ?

We have seen many searching strategies till now, but as we can see no one is really the perfect. How can we make our AI agent to search better ?

We can make use of a concept called as **metalevel state space.** Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an object-level state space such as in Indian Traveller Problem.

Consider A* algorithm

1) In A* algorithm it maintains internal state which consists of the current search tree.
2) Each action in the metalevel state space is computation step that alters the internal state.

For example - Each computation step in A* expands a leaf node and adds its successors to the tree.

3) As the level increases the sequence of larger and larger search trees is generated.

In harder problem there can be mis-steps taken by algorithm. That is algorithm can explore unpromising unuseful subtrees. Metalearning algorithm overcomes these problems.

The main goal of metalearning is to minimize the total cost of problem solving.

3.1.10 Heuristic Functions and Their Nature

3.1.10.1 Accuracy of Heuristic Function

- More accurate the heuristic function more is the performance.
- The quality of heuristic function can be measured by the effective branching factor b*.
- Consider A* that generates N nodes and 'd' is depth of a solution, then b* is the branching factor that a uniform tree of depth 'd' would have so as to contain 'n+1' nodes.

Thus,

$$N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

If A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

A well designed heuristic function will have a value of b* close to 1, allowing fairly large problems to be solved.

3.1.10.2 Designing Heuristic Functions for Various Problems

Example 3.1.1 *8-puzzle problem*

Solution : The objective of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.

7	2	4
5		6
8	3	1

Start state

	1	2
3	4	5
6	7	8

Goal state

1) The average solution cost for a randomly generated 8-puzzle instance is about 22 steps.

2) The branching factor is about 3 (when the empty tile is in the middle, there are four possible moves, when it is in a corner there are two, and when it is along an edge there are three.

3) An exhaustive search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states.

4) If we keep track of repeated states, we could cut this down by a factor of about 1,70,000, because there are only 9!/2 = 1,81,440 distinct states that are reachable.

5) If we use A*, we need a heuristic function that never overestimates the number of steps to the goal.

6) We can use following heuristic function,

- h_1 = The number of misplaced tiles. All of the eight tiles are out of position, so the start state would have $h_1 = 8 \cdot h_1$, is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

- h_2 = The sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances.

Example 3.1.2 *Blocks world problem.* **GTU : Winter-17, Marks 3**

Solution :

Heuristic for Blocks World

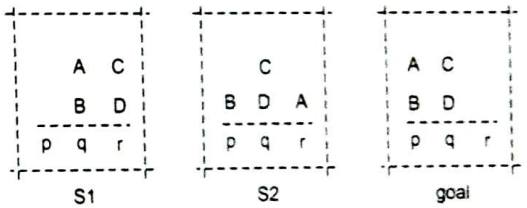
A function that estimates the cost of getting from one place to another (from the current state to the goal state.)

Used in a decision process to try to make the best choice of a list of possibilities (to choose the move more likely to lead to the goal state.)

Best move is the one with the least cost.

The "intelligence" of a search process, helps in finding a more optimal solution.

h1(s) = Number of places with incorrect block immediately on top of it.



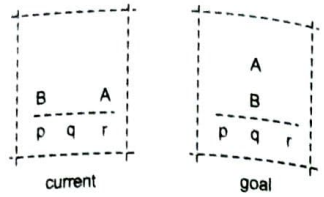
h1 (S1) = 3 h1 (S2) = 1

A more informed heuristic

Looks at each bottom position, but takes higher positions into account. Can be broken into 4 different cases for each bottom position.

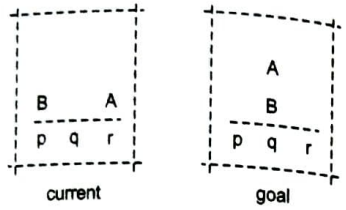
1. Current state has a blank and goal state has a block.

Two blocks must be moved onto q. So increment the heuristic value by the number of blocks needed to be moved into a place.



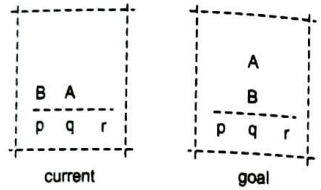
2. Current state has a block and goal state has a blank.

One block must be removed from p. So increment the heuristic value by the number of blocks needed to be moved out of a place.



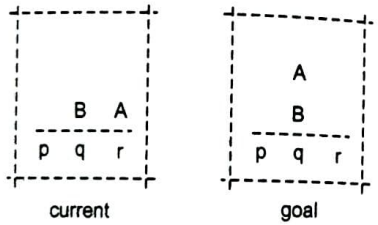
3. Both states have a block, but they are not the same.

On place q, the blocks don't match. The incorrect block must be moved out and the correct blocks must be moved in. Increment the heuristic value by the number of these blocks.

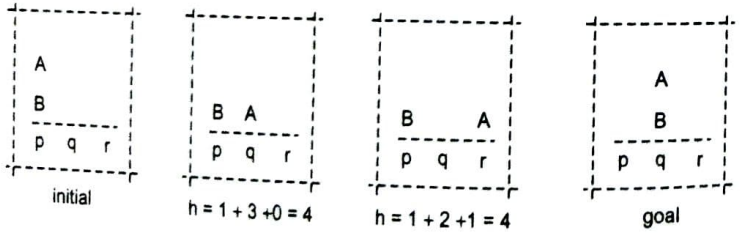


4. Both states have a block, and they are the same.

B is in the correct position. However, the position above it is incorrect. Increment the heuristic value by the number of incorrect positions above the correct block.



Example



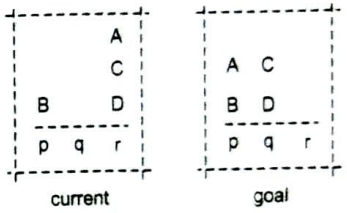
Know the second state leads to the optimal solution, but not guaranteed to be chosen first. Can the heuristic be modified to decrease its value? Yes, since the block on p can be moved out and put on q in the same move. Need to compensate for this overcounting of moves.

For a given (nonempty) place, find the top-most block. Look for its position in the goal state. If it goes on the bottom row and the place is currently empty, the block can be moved there, so decrement heuristic by one.

Now the second state's heuristic will be 3, and it's guaranteed to be chosen before the other one. A little like a one move look ahead.

If the top-most block doesn't go on bottom row, but all blocks below it in the goal state are currently in their correct place, then the block can be moved there, so decrement heuristic by one.

Example



p: case 4: +1

q: case 1: +2

r: case 2: +3

A can be put on B: -1

so h=5

optimal number of moves is 4

Example 3.1.3 *Travelling salesman problem.* **GTU : Summer-18, Marks 3**

Solution : Heuristic function, h(n) is defined as,

h(n) = estimate cost of the cheapest path from node 'n' to a goal state.

Heuristic function for Travelling salesman problem :

- Travelling salesman problem (TSP) is a routing problem in which each city must be visited exactly once. The aim is to find the shortest tour.
- The goal test is, all the locations are visited and agent at the initial location.
- The path cost is distance between locations.
- As a heuristic function for TSP, we can consider the sum of the distances travelled so for. The distance value directly affects the cost therefore, it should be taken into calculation for heuristic function.

Heuristic function for 8 puzzle problem.

- The objective of the 8 puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.
- We can use following heuristic function,
  - h_1 = The number of misplaced tiles. All of the eight tiles are out of position, so the start state would have $h_1 = 8 \cdot h_1$, is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.
  - h_2 = The sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances.

Example 3.1.4 *Tic Tac Toe problem.* **GTU : Summer-18, Marks 3**

Solution :

(ii) Heuristic for Tic Tac Toe problem

This problem can be solved all the way through the minimax algorithm but if a simple heuristic/evaluation function can help save that computation. This is a static evaluation function which assigns a utility value to each board position by assigning weights to each of the 8 possible ways to win in a game of tic-tac-toe and then summing up those weights.

One can evaluate values for all of 9 fields, and then program would choose the field with highest value.

For example,

Every Field has several WinPaths on the grid. Middle one has 4 (horizontal, vertical and two diagonals), corners have 3 each (horizontal, diagonal and one vertical), sides have only 2 each (horizontal and vertical). Value of each Field equals sum of its WinPaths values. And WinPath value depends on its contents:

- Empty : [| |] - 1 point
- One symbol: [X| |] - 10 points // can be any symbol in any place
- Two different symbols: [X|O|] - 0 points // they can be arranged in any possible way
- Two identical opponents symbols: [X|X|] - 100 points // arranged in any of three ways
- Two identical "my" symbols: [O|O|] - 1000 points // arranged in any of three ways
- This way for example beginning situation has values as below :

3	2	3
2	4	2
3	2	3

3.1.10.3 The Domination of Heuristic Function

- If we could design multiple heuristic function for same problem then we can find that which one is better.
- Consider two heuristic functions h_1 and h_2. From their definitions for some node n, if $h_2(n) \geq h_1(n)$ then we say that h_2 dominates h_1.
- Domination directly points to accuracy . A* using h_2 will never expand more nodes than A* using h_1 (except for some node having $f(n) = c^*$)

3.1.10.4 Admissible Heuristic Functions

1) A problem with fewer restrictions on actions is called a **relaxed problem.**
2) The cost of an admissible solution to a relaxed problem is an admissible heuristic for the original problem. The heuristic function is admissible because the optimal solution in the original problem is, also a solution in the relaxed problem, and therefore must be at least as expensive as the optimal solution in the relaxed problem.
3) As the derived heuristic is an exact cost for the relaxed problem, therefore it is consistant.
4) If problem definitions are written in **formal** languages then it is possible to construct relaxed problem automatically.
5) Admissible heuristic function can also be derived from the solution cost of a subproblem of the given problem. The cost of optimal solution of this subproblem would be the lower bound on the cost of the complete problem.
6) We can store the exact solution costs for every possible subproblem instance in a database. Such a database is called as **pattern database.**

   The pattern database is constructed by searching backwards from the goal state and recording the cost of each new pattern encountered.

   We can compute an admissible heuristic function 'h' for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database.
7) The cost of the entire problem is always greater than sum of cost of two subproblems. Hence it is always better to derive disjoint solution and then sum up all solutions to minimize the cost.

3.1.11 Learning Heuristic from Experience

1) A heuristic function h(n) is supposed to estimate the cost of a solution beginning from the state at node n. Therefore it is really difficult to design h(n).
2) One solution is to devise relaxed problems for which an optimal solution can be found.
3) Another solution is to make agent program that can learn from experience.
   - "Learn from experience" means solving similar problem again and again (i.e. practicing).
   - Each optimal solution will provide example from which 'h(n) design' can be learned.
   - One can get experience of which h(n) was better.
   - Each example consists of a state from the solution path and the actual cost of the solution from that point.
   - From such solved examples an inductive learning algorithm can be used to construct the function h(n) that can predict solution costs for another states that have arised during search. [A lucky agent program will get the prediction early.]
   - For developing inductive algorithms we can make use of techniques like neural nets, decision trees, etc.
   - If inductive learning methods have knowledge about features of a state that are relevant to evaluation of algorithms then inductive learning methods give best output.

3.2 Local Search Algorithms and Optimization Problems

GTU : Summer-12,13,14,15,16,17,18,19,20, Winter-12,14,15,16,17,18,19

- The search algorithms we have seen so far, more often concentrate on path through which the goal is reached. But if the problem does not demand the path of the solution and it expects only the final configuration of the solution then we have different types of problem to solve.
- Following are the problems where only solution state configuration is important and not the path which has arrived at solution.
  1) 8-queen (where only solution state configuration is expected).
  2) Integrated circuit design. 3) Factory-floor layout.
  4) Job-shop scheduling. 5) Automatic programming.
  6) Telecommunication network optimization.

  7) Vehicle routing. 8) Portfolio management.

If we have such type of problem then we can have another class of algorithms, the algorithms that do not worry about the paths at all. These are local search algorithms.

3.2.1 Local Search Algorithms

- They operate using single state. (rather than multiple paths).
- They generally move to neighbours of the current state.
- There is no such requirement of maintaining paths in memory.
- They are not "Systematic" algorithm procedure.
- The main advantages of local search algorithm are

1) They use very little and constant amount of memory.
2) They have ability to find resonable solution for infinite state spaces (for which systematic algorithms are unsuitable).

Local search algorithms are useful for solving **pure optimization problems.** In pure optimization problems main aim is to find the best state according to required objective function.

Local search algorithm make use of concept called as **state space landscape.** This landscape has two structures -

1) Location (defined by the state)
2) Elevation (defined by the value of the heuristic cost function or objective function).

- If elevation corresponds to the cost, then the aim is to find the **lowest valley - (a global minimum).**
- If elevation corresponds to the objective function then aim is to find the **highest peak - (a global maximum).**
- Local search algorithms explore the landscape.

Performance measurement

- Local search is complete i.e. it surely finds goal if one exists.
- Local search is optimal as it always find global minimum or maximum.

Local search representation

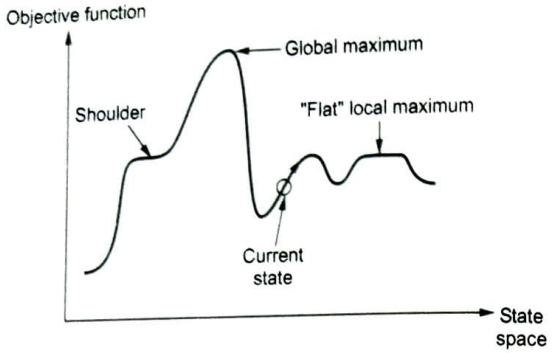


Fig. 3.2.1 Local search

3.2.2 Hill Climbing Search

- This algorithm generally moves up in the direction of increasing value that is-uphill. It breaks its "moving up loop" when it reaches a "peak" where no neighbour has a higher value.
- It does not maintain a search tree. It stores current node data structure. This node records the state and its objective function value. Algorithm only look out for immediate neighbours of current state.
- It is similar to greedy local search in a sense that it considers a current good neighbour state without thinking ahead.
- Greedy algorithm works very well as it is very easy to improve bad state in hill climbing.

3.2.2.1 Algorithm for Hill Climbing

The algorithm for hill climbing is as follows : -

1) Evaluate the initial state. If it is goal state quit, otherwise make current state as initial state.
2) Select a new operator that could be applied to this state and generate a new state.
3) Evaluate the new state. If this new state is closer to the goal state than current state make the new state as the current state. If it is not better, ignore this state and proceed with the current state.
4) If the current state is goal state or no new operators are available, quit. Otherwise repeat from 2.

3.2.2.2 Problems with Hill Climbing

1) Local maxima - can't see higher peak.
2) Shoulder - can't see the way out.

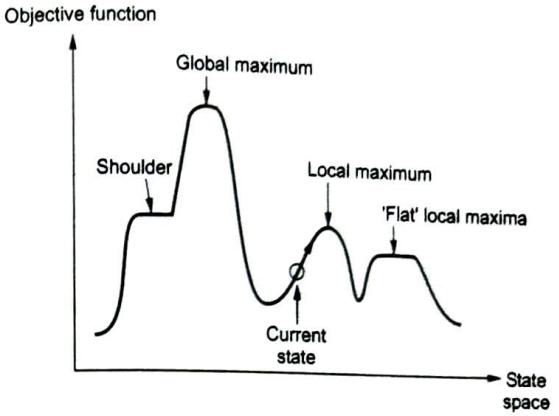


Fig. 3.2.2 Various stages in hill climbing search

Local maxima - It is a state where we have climbed to the top of the hill, and missed on better solution.

It is the mountain - A state that is better than all of its neighbours, but not better than some other states further away. **[Shown in Fig. 3.2.3]**

Fig. 3.2.3 Local maxima

Plateau : It is a state where everything around is about as good as where we are currently. In other words a flat area of the search space in which all neighbouring states have the same value. **[Shown in Fig. 3.2.4]**

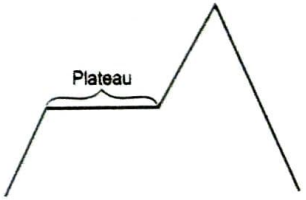


Fig. 3.2.4 Plateau

Ridges : In this state we are on a ridge leading up, but we can't directly apply an operator to improve the situation, so we have to apply more than one operator to get there. **[Shown in Fig. 3.2.5]**

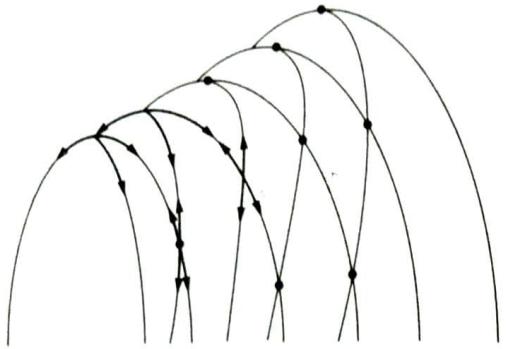


Fig. 3.2.5 Ridges

Illustration of ridges : The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequences of local maxima that are not directly connected to each other. From each local maximum all the available actions point downhill.

3.2.2.3 Solving Problems Associated with Hill Climbing

- All the above discussed problems could be solved using methods like backtracking, making big jumps (to handle plateaus or poor local maxima), applying multiple rules before testing (helps with ridges) etc. Hill climbing is best suited to problem where the heuristic gradually improves, the closer it gets to the solution; it works poorly where there are sharp drop-offs. It assumes that local improvement will lead to global improvement.

3.2.2.4 Example for Local Search

Consider the 8-queens problem :

A complete-state formulation is used for local search algorithms. In 8-queens problem, each state has 8-queens on the board one per column. There are two functions related with 8-queens.

1) The successor function : It is function which returns all possible states which are generated by a single queen move to another cell in the same column. The total successor of the each state $8 \times 7 = 56$.

2) The heuristic cost function : It is a function 'h' which hold the number of attacking pair of queens to each other either directly or indirectly. The value is zero for the global minimum of the function which occurs only at perfect solutions.

3.2.2.5 Advantages of Hill Climbing

- Hill climbing is an optimization technique for solving computationally hard problems.
- It is best used in problems with the property that the" state description itself contains all the information needed for a solution".
- The algorithm is memory efficient since it does not maintain a search tree. It looks only at the current state and immediate future states.
- In contrast with other iterative improvement algorithms, hill-climbing always attempts to make changes that improve the current state. In other words, hill-climbing can only advance if there is a higher point in the adjacent landscape.
- It is often useful when combined with other methods, getting it started right in the immediate general neighbourhood.

3.2.2.6 Variations of Hill Climbing

Many variants of hill-climbing have been invented as discussed below.

1) Stochastic hill climbing : Chooses at random from among the uphill moves ; the probability of selections can vary with the steepness of the uphill move.

2) First choice hill climbing : Implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g. thousands) of successors.

3) Random restart hill climbing : Adopts the well known saying "If at first you don't succeed, try, try again". It conducts a series of hill climbing searches from randomly generated initial states, stopping when a goal is found.

The hill climbing algorithms described so far are incomplete because they often fail to find a goal when surely goal exist. This can happen because these algorithms can get stuck on local maxima. Random restart hill is complete with probability approaching to 1. This algorithm do not stop until it reaches to goal.

The success of hill climbing depends very much on the shape of the state-space landscape. If there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.

4) Steepest Ascent Hill Climbing : This algorithm differs from the basic Hill climbing algorithm by choosing the best successor rather than the first successor that is better. This indicates that it has elements of the breadth first algorithm.

Steepest ascent Hill climbing algorithm

1. Evaluate the initial state

2. If it is goal state then quit otherwise make the current state this initial state and proceed;
3. Repeat set target to be the state that any successor of the current state can better; for each operator that can be applied to the current state apply the new operator and create a new state evaluate this state.
4. If this state is goal state Then quit. Otherwise compare with Target. If better set Target to this value. If Target is better than current state set current state to Target. Until a solution is found or current state does not change.

Both the basic and this method of hill climbing may fail to find a solution by reaching a state from which no subsequent improvement can be made and this state is not the solution.

Local maximum state is a state which is better than its neighbours but is not better than states faraway. These are often known as foothills. Plateau states are states which have approximately the same value and it is not clear in which direction to move in order to reach the solution. Ridge states are special types of local maximum states. The surrounding area is basically unfriendly and makes it difficult to escape from, in single steps, and so the path peters out when surrounded by ridges. Escape relies on: backtracking to a previous good state and proceed in a completely different direction-involves keeping records of the current path from the outset; making a gigantic leap forward to a different part of the search space perhaps by applying a sensible small step repeatedly, good for plateau; applying more than one rule at a time before testing, good for ridges. None of these escape strategies can guarantee success.

3.2.3 Simulated Annealing Search

1) In simulated annealing, initially the whole space is explored.
2) This is a variation of hill climbing.
3) This avoids the danger of being caught on a plateau or ridge and makes the procedure less sensitive to the starting point.
4) Simulated annealing is done to include a general survey of the scene, to avoid climbing, false foot hills.
5) There are two additional changes -
   i) Rather than creating maxima, minimisation is done.
   ii) The term objective function is used rather than heuristic.
6) This concept is taken from physical annealing where physical substances are melted and then gradually cooled until some solid state is reached. In physical annealing the goal is to produce a minimal-energy state. The annealing schedule

states, that if the temperature is lowered sufficiently slowly, then the goal will be attained. ΔE is called the change in the value of the objective function.

7) It becomes clear that in this algorithm we have valley descending rather than hill climbing. The probability that the metal will jump to a higher energy level is given by $P = \exp^{8-E/kT}$ where k is Boltzmann's constant.

The algorithm

1) Start with evaluating the initial state.
2) Apply each operator and loop until a goal state is found or till no new operators left to be applied as described below : -
   i) Set T according to an annealing schedule.
   ii) Select and apply a new operator.
   iii) Evaluate the new state. If it is a goal state quit.

   ΔE = Val (current state) – Val (new state)

- If $\Delta E < 0$ then this is the new current state.

Else find a new current state with probability $e - E / kT$.

3.2.4 Local Beam Search

1) In the local beam search, instead of single state in the memory k states are kept in the memory.
2) In local beam search states are generated in random fashion.
3) A successor function plays an important role by generating successor of all K states.
4) If any one successor state is goal then no further processing is required.
5) In other case i.e. if goal is not achieved, it observes the K best successors from the list of all successor and process is repeated.
6) At first glance the random parallism is achieved in the sequence by a local beam search with K states.
7) In a random-restart search every single search activity run independently of others.
8) To implement local search, threads are used. The K parallel search threads carry useful information.
9) The algorithm work on principle of successful successors. If one state generate efficient/goal reaching successor and other K-1 state generate poor successor, then in this situation the successful successors leads all other states.
10) The algorithm drops/leaves the unsuccessful search and concentrate on successful search.

Limitation of local beam search

1) The local beam search has limitation of, lack of variation among the K states.
2) If the state concentrate on small area of state space then search becomes more expensive.

3.2.5 Stochastic Beam Search

1) Its one of the flavour of local beam search, which is very similar to that of stochastic hill climbing.
2) It resolves the limitation of local beam search.
3) Stochastic beam search concentrate on random selection of K successor instead of selecting k best successor from candidate successor.
4) The probability of random selection is increasing function of its success rate.
5) A stochastic beam search is very similar to natural selection, where the child (successor) of a parent state is eugenic (good production) according to its success rate (fitness). Thus the next generation is also very much powerful.

3.2.6 Genetic Algorithms

1) Evolutionary pace of learning algorithm is genetic algorithm. The higher degree of eugency can be achieved with new paradigm of AI called a genetic algorithms.
2) A genetic algorithm is a rich flavour of stochastic beam search.
3) In the genetic algorithm two parent states are combined together by which a good successor state is generated.
4) The analogy to natural selection is the same as in stochastic beam search, except now we are dealing with sexual rather than asexual reproduction.

3.2.6.1 Term used in Genetic Algorithm

1) Population : Population is set of states which are generated randomly.

2) Individual : It is a state or individual and it is represented as string over a finite alphabet.

Example - A string of 0s and 1s.

For example - In 8-queen all states are specified with the position of 8-queens. The memory required is

$8 \times \log_2 8 = 24$ bits. Each state is represented with 8 digits.

3) Fitness function : It is a evaluation function. On its basis each state is rated. A fitness function should return higher values for better states. For the state, the probability of being chosen for reproducing is directly proportional to the fitness score.

In 8-queen problem the fitness function has 28 value for number of nonattacking pairs.

4) Crossover : Selection of state is dependent on fitness function. If fitness function value is above threshold then only state is selected otherwise discarded. For each state, pairs are divided, that division point or meeting point is called crossover point, which is chosen in random order from the positions in the string.

5) Mutation : Mutation is one of the generic operator. Mutation works on random selections or changes. For example mutation select and changes a single bit of pattern switching 0 to 1 or 1 to #.

6) Schema : The schema is a substring in which position of some bit can be unspecified.

3.2.6.2 Working of a Genetic Algorithm

Input : 1) State population (a set of individuals)
2) Fitness function (that rates individual).

Steps

1) Create an individual 'X' (parent) by using random selection with fitness function 'A' of 'X'.
2) Create an individual 'Y' (parent) by using random selection with fitness function 'B' of 'Y'.
3) Child with good fitness is created for **X + Y.**
4) For small probability apply mutate operator on child.
5) Add child to new population.
6) The above process is repeated until child (an individual) is not fit as specified by fitness function.

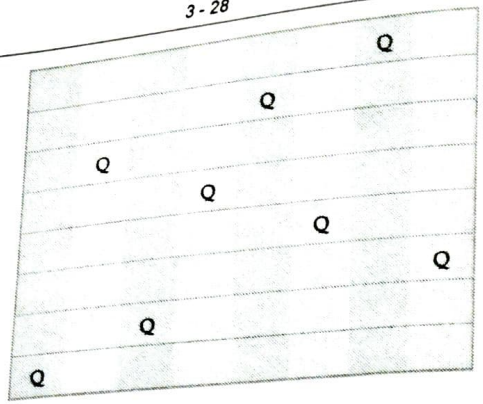
Genetic algorithm example

Example : 8-queens problem states.

States :

Assume each queen has its own column, represent a state by listing a row where the queen is in each column (digits 1 to 8).

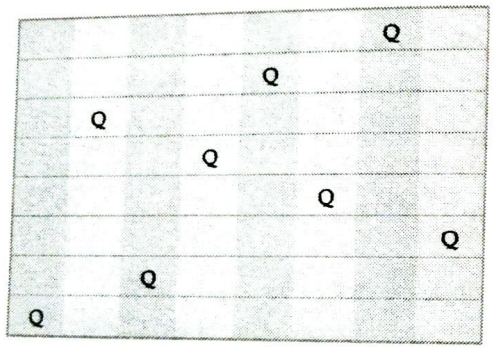
For example, the state below will be represented as 16257483.



3.2.6.3 Example : 8 Queens Problem Fitness

Fitness function : Instead of – h as before, use the number of non-attacking pairs of queens. There are 28 pairs of different queens, smaller column first, all together, so solutions have fitness 28. (Basically, fitness function is 28 – h)

For example, fitness of the state below is 27 (queens in columns 4 and 7 attack each other).



Example : 8 queens problem crossover.

Choose pairs for reproduction (so that those with higher fitness are more likely to be chosen, perhaps multiple times).

For each pair, choose a random crossover point between 1 to 8, say 3.

Produce offspring by taking substring 1-3 from the first parent and 4 - 8 from the second (and vice versa). Apply mutation (with small probability) to the offspring.

Importance of representation

Parts we swap in crossover should result in a well-informed solution (and in addition better be meaningful).

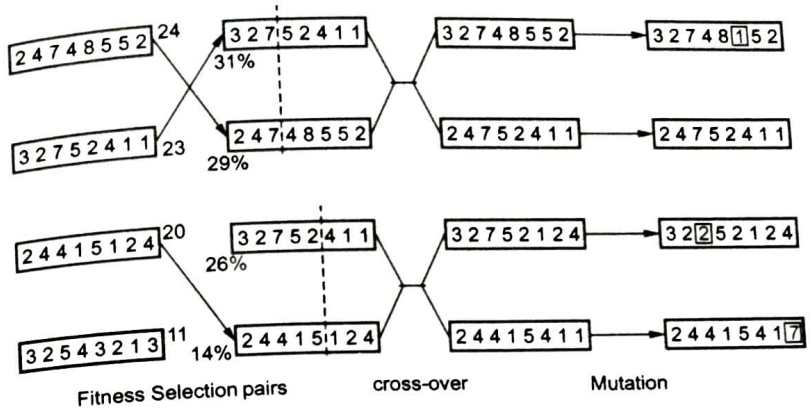
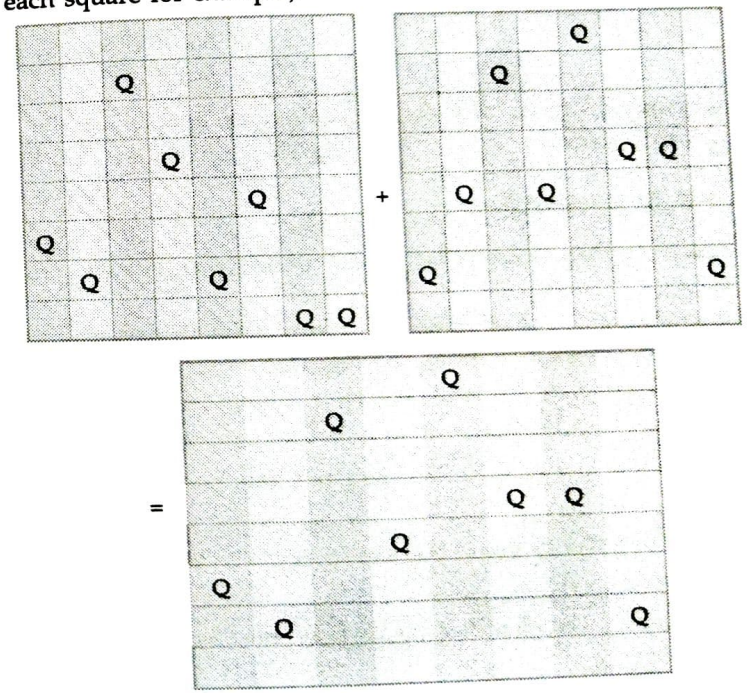


Fig. 3.2.6 8 Queen problem crossover

Consider what would happen with binary representation (where position requires 3 digits).

Also chosen representation reduces search space considerably (compared to representing each square for example).



3.2.6.4 Optimization using Genetic Algorithm

1) Genetic algorithm is biological model of intelligent evolution. It generates the population of competing children.
2) The poor candidate solution will be vanished, as genetic algorithm generates best child.

Thus, in practice genetic algorithm is most promising survive and reproduction technique by constructing new optimized solution. Thus it is optimization oriented technique.

Application of genetic algorithm on optimization.

1) Circuit layout.
2) Job-shop scheduling.

3.3 Local Search in Continuous Spaces

As we know that environment can be descrete or continuous, but the most real-world environment are continuous.

1) It is very difficult to handle continuous state space. The successor for real-world problem are many infinite states.
2) Origin of local search in continuous spaces lies in Newton and Leibnitz in the 17^{th} century.
3) Optimal solution for given problem in continuous spaces can be found with "Local search techniques".

3.3.1 Search and Evaluation Theory

1) Search is basic problem solving technique. Basically it is always related with evolution theory.
2) Charles Darwin is father of evolutionary theory. The theory was based on the origin of species by means of natural selection (1859).
3) The variations (mutations) are well known attributes of reproduction and best features are preserved in next generation in proper propagation.
4) The qualities are inherited or modified. This fact was not associated with Darwin theory.
5) Gregor Mendel (1866) theory found the fact of inheritance. He performed artificial fertilization on peas.
6) DNA module structure was identified by Watson and Crick.

   A → Adenin, G - Guanine,

   T → Thymine, C - Cytosine

7) The key difference between stochastic beam search and evolution is that successors are generated from multiple organisms (states) rather than one organism (state).
8) The theory of evolution is very much rich than genetic algorithm.
9) The process of mutations involves duplication, reversals and motion of large group of DNA.
10) Most important is the fact that the genes themselves encode the mechanisms whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.
11) French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits acquired by adaptation during an organism's lifetime would be passed on to its offspring. Such a process would be effective, but does not seem to occur in nature.
12) James Baldwin (1896) proposed a superficially similar theory : - that behavior learned during an organism's lifetime could accelerate the rate of evolution.

For example -

Suppose we want to place **three new airports anywhere in India,** such that the sum of squared distances from each city to its nearest airport is minimized.

i) The state space, is defined by the co-ordinates of the airports :

   (x_1, y_1), (x_2, y_2) and (x_3, y_3).

ii) This is a six-dimensional space : We also say that states are defined by six variables. (In general, states are defined by an n-dimensional vector of variables, x).

iii) Moving around in this space corresponds to moving one or more of the airports on the map.

iv) The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state, once we compute the closest cities, but rather tricky to write down in general.

3.3.2 Problems Associated with Local Search

- Local search methods suffer from local maxima, ridges and plateaux in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing can be used and are often helpful. High-dimensional continuous spaces are, however, big places in which it is easy to get lost.

3.3.3 Constrained Optimization Problem

- An optimization problem is constrained if solutions must satisfy some hard constraint like sites to be inside India and on dry land (rather than in the middle of rivers). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of linear programming problems, in which constraints must be linear inequalities forming a convex region and the objective function is also linear. Linear programming problems can be solved in time which is polynomial in the number of variables.

3.4 Handling Unknown Environments

- Earlier algorithms we have seen are offline search algorithms. They compute a complete solution before setting foot in the real world, and then execute the solution without considering new percepts. Now we will consider how to handle dynamic or semidynamic environments.

3.4.1 Online Search Agents

1) The online search agent works on the concept of interleaving of two tasks namely computation and action. In the real-world, the agent perform an action and it observes the environment and then work out on the next action.
2) If an environment is dynamic or semidynamic or stochastic the online search work in best manner.
3) An agent needs to pay a penalty for lengthy computation and for sitting around.
4) An offline search is costly. For offline search we need to have proper planning which consider all related information. In online search there is no need of planning just see what happens and act accordingly.

   Example - Chess moves.
5) In online search the states and actions are unknown to the agent. Thus it has an exploration problem. (it needs to explore the world).
6) In an online search the actions are used to predict next states and actions.
7) For predicting next states and actions computations are performed.

For example -

- Consider example of robot who builds a map between 2 locations say LOC 1 and LOC 2. For building a map robot needs to explore the world so as to reach to LOC 2 from LOC 1. Parallely it will build the map.

- A natural real world example of exploration problem and online search is a baby trying to understand the world around it. It is baby's online search process. A baby tries an action and go in certain state. It keeps on doing this, gathering new experience and learning from it.

3.4.2 Solving Online Search Problems

1) The purely computation process cannot handle an online search problem properly. Agent needs to execute actions for solving problem.
2) The agent has knowledge about -
   a) ACTIONS (s) which returns a list of actions allowed in state s.
   b) The step-cost function cost (s_1, a, s') (note that cost function cannot be used until the agent knows that s' is the outcome).
   c) GOAL-TEST (s)
3) It is not possible for agent to access the successors of a state. In fact the agent try related actions, in that state.
4) As basic objective of an agent is to minimize cost, the goal state must be found with minimum cost.
5) The another possible goal can be to explore the entire environment. The total path cost is cost of path across which the agent travels.
6) In the online algorithm we can find the competitive ratio means shortest path. The competitive ratio must be very small. But in reality many a time competitive ratio is infinite.
7) If the online search is with irreversible action then it will reach to a dead-end state and from that state it can't achieve goal state.
8) We must build an algorithm which does not explore dead-end path. But accidently or in reality an algorithm can avoid dead ends in all state spaces.
9) Consider two goal state space as shown in Fig. 3.4.1.

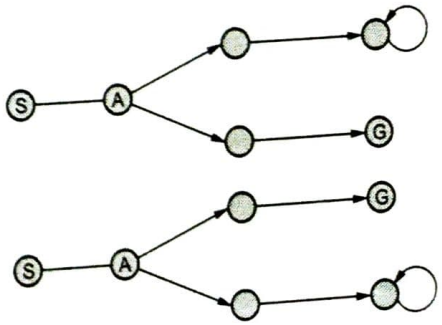


Fig. 3.4.1 Two goal state space which may lead to a dead end

In the Fig. 3.4.1 online search algorithm visit states S and A, both the state spaces are identical so it must be explored in both cases. One link is to goal state (G) and other lead to dead end in both states spaces. It is an

example of an adversary arguments. The exploration of state space is in an adversary manner. One can put the goals and dead ends likewise.

3.4.3 Online Local Search

Hill climbing search

1) Hill climbing search has the property of locality in its node expansions because it keeps just one current state in memory, hill-climbing search is already an online search algorithm.
2) Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go.
3) Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Random walk

1) Instead of random restarts, one might consider using a random walk to explore the environment.
2) A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried.
3) It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite.

Hill climbing with memory

1) Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach.
2) The basic idea is to store a "current best estimate", H(S) the cost to reach the goal from each state that has been visited.
3) H(S) starts out being just the heuristic estimate h(s) and is updated as the agent gains experience in the state space.

3.4.4 Learning Real Time A*

1) An agent can use a scheme called learning real-time A* (LRTA*).
2) It builds a map of the environment using the result table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates.
3) One important detail is that actions that have not yet been tried in a state S are always assumed to lead immediately to the goal with the least possible cost,

namely h(s). This optimism under uncertainty encourages the agent to explore new, possibly promising paths.

4) An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment.
5) It is not complete for infinite state spaces. There are cases where it can be led infinitely astray. It can explore an environment of n states in $O(n^2)$ steps in the worst case.

3.4.5 Learning in Online Search

At every step online search agent can learn various aspects of current situation.

The agents learn a "map" of the environment - more precisely, the outcome of each action in each state-simply by recording each of their experiences.

The local search agents acquire more accurate estimates of the value of each state by using local updating rules.

By learning, the agent can do better searching.

3.5 Constraint Satisfaction Problems

GTU : Summer-12,13,14,16,18, Winter-14,15

- Constraint satisfaction problems are problems whose states and goal test conform to a standard, structured and very simple representation. Search algorithm can be defined that take advantage of the structure of states and can use general-purpose rather than problem-specific heuristics to enable the solution of large problems.

3.5.1 Constraint Satisfaction Problems - The Concept

1. Constraint satisfaction problem has various states and goal test, a tranditional problem has been converted into standard structured and very simple "representation.".
2. The general-purpose routines can be used to access a special representation which has more benefit than problem-specific heuristics. These routines combined with special structure can find solution of large problems.
3. The structure of the problem is represented in different form such as, standard representation of the goal test.
4. The revealed structured is very efficient in many ways such as,
   i) Problem decomposition.
   ii) To understand structure of problem and the diffculty of solving it and their connection.

5. If the problem is treated as CSP we have many advantages, as discussed below.

   i) As the representation of states have standard pattern [that is a set of variables with assigned values], we can design successor function and goal test in generic way that will apply to all CSPs.

   ii) We can develop effective generic heuristic that require no additional domain specific expertise.

   iii) The structure of the constraint graph can be used to simplify the solution process in some cases giving exponential reduction in complexity.

3.5.2 Formal Definition of Constraint Satisfaction Problems

1. A constraint satisfaction problem is defined by a set of variables $X_1, X_2, ..., X_n$ and a set of constraints $C_1, C_2, ..., C_m$.
2. Each variable X_1 has a nonempty domain D_1 of possible values.
3. Each constraint C_1 involves some subset of the variables and specifies the allowable combinations of values for that subset.
4. A state of the problem is defined by an assignment of values to some or all of the variables.
   $\{X_i = V_i, X_j = V_j, ...\}$.
5. An assignment that does not violate any constraints is called a consistent or legal assignment.
6. A complete assignment is one in which every variable is mantained and a solution to a CSP is, a complete assignment that satisfies all the constraints.
7. Some CSPs also require a solution that maximizes an objective function.

Consider graph colouring problem as shown in Fig. 3.5.1. Constraints are -

1. We have three colours for colouring a vertex.
2. No two adjacent vertices have same colour.

Given three colours-(Red, Green, Blue). Allowable combination for A, B vertices would be,

{(R, G), (R, B), (G, R), (G, B), (B, R), (B, G)}

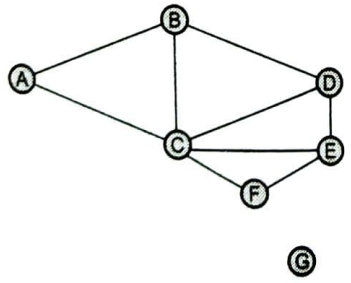


Fig. 3.5.1 Graph colouring problem

3.5.3 Examples of CSP

3.5.3.1 Map Colouring Problem

1. The problem is to colour the regions of a given map such that no 2 adjacent regions have the same colour. [Refer figure 3.5.2]
2. The regions in the map are the variables and the set of possible colours for the regions is the domain.
3. The constraint is that " no two adjacent regions should have the same colour."

Formal Representation of Map Colouring Problem

1. **Variables :** {N, NW, NE, M, MW, ME, S, SE}
2. **Domains :** D = {red, green, blue}
3. **Constraints :** Adjacent regions must have different colors.

Example : N ≠ NW

Note :

1. Typically, such a problem has many solutions.
2. We sometimes represent map colouring as a graph coloring (constraint graph) problem.
3. The topology of a constraint graph can sometimes be used to identify solutions easily.

Example Map (See Fig. 3.5.2 on next page).

3.5.3.2 Other Examples of CSP

1. N-queens puzzle.
2. Jobshop scheduling.
3. Scene labelling.
4. Circuit board layout.
5. Map colouring problem.
6. Sudoku
7. Boolean satisfiability.

Some real-world problems -

1. Assignment problems.
2. Transporation scheduling.
3. Hardware configuration.
4. Spreadsheets.
5. Factory scheduling.
6. Floor planning.

3.5.4 Incremental Formulation for CSP

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

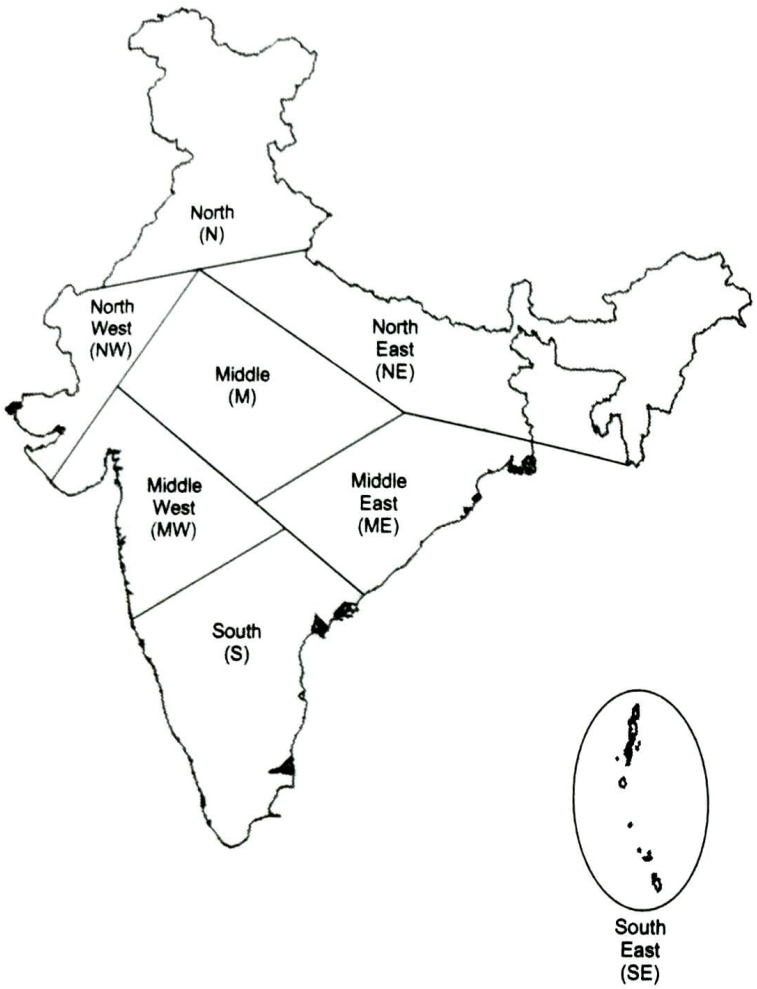


Fig. 3.5.2 Map colouring problem

i) Initial state -

The empty assignment {}, in which all variables are unassigned.

ii) Successor functions -

A value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.

iii) Goal test -

The current assignment is complete.

iv) Path cost -

A constant cost for every step.

3.5.5 Searching for Goal State in CSP

1. Every solution must be a complete assignment and therefore appears at depth n if there are n variables.
2. The search tree extends only to depth n.
3. Depth-first search algorithms are popular for CSPs.
4. The path by which a solution is reached is irrelevant.
5. We can also use a complete-state formulation, in which every state is a complete assignment that might or might not satisfy the constraints.
6. Local search methods work well for this formulation.

3.5.6 Variations in CSPs

1. The simplest kind of CSP involves variables that are discrete and have finite domains. Graph-coloring problems are of this kind. The 8-queens problem described can also be viewed as finite-domain CSP, where the variables $Q_1, \ldots, Q_8$ are the positions of each queen in colums 1, ..., 8 and each variable has the domain {1, 2, 3, 4, 5, 6, 7, 8}. If the maximum domain size of any variable is a CSP in d, then the number of possible complete assignments are O (d^n) that is exponential in the number of variables.
2. Finite-domain CSPs include Boolean CSPs, whose variables can be either true or false. Boolean CSPs include, as special cases, some NP-complete problems, such as 3SAT.
3. In most practical applications, however, general-purpose CSP algorithms can solve problems of orders of magnitude larger than those solvable via the general-purpose search algorithms that we saw.
4. Discrete variables can also have infinite domains-for example, the set of integers or the set of strings.
5. Constraints satisfaction problems with continuous domains are very common in the real world and are widely studies in the field of operations research.

   For example, the scheduling of experiments on the Hubble Space Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints. The best-known

category of continuous-domain CSPs is that of linear programming problems where constraints must be linear inequalities.

3.5.7 Constraints in CSPs

3.5.7.1 Properties of Constraints

Constraints are used to guide reasoning of everyday common sense. The constraints have following properties.

1. Constraints may specify partial infomation; constraint need not uniquely specify the values of its variables.
2. Constraints are non-directional, typically a constraint on (say) two variables V_1, V_2 can be used to infer a constraint on V_1 given a constraint on V_2 and vice versa ;
3. Constraints are declarative ; they specify what relationship must hold without specifying a computational procedure to enforce that relationhip.
4. Constraints are additive; the order of imposition of constraints does not matter, all that matters at the end is that the conjunction of constraints is in effect.
5. Constraints are really independent; typically constraints in the constraint store (i.e. collection of constraints) share variables.

3.5.7.2 Types of Constraints in CSPs

1. **Unary constraint -**

   Which restricts the value of a single variable. Every unary constraint can be eliminated simply by prepressing the domain of the corresponding variable to remove any value that violates the constraint.

   For example - Constraint can be that, Vertex A can not be coloured with blue colour.

2. **Binary constraint -**

   Relates two variables. A binary CSP is one with only binary constraints; it can be represented as a constraint graph.

   For example - In graph colouring problem two adjacent vertices can not have same colour.

3. **Higher-order constraints -**

   Involves three or more variables. A familiar example is provided by cryptarithmetic puzzles. It insist that each letter in a cryptarithmetic puzzle represent a different digit. Higher-order constraints can be represented in a constraint hypergraph. Such as shown below : -

For example - The cryptarithmatic problem. (A CSP problem having high order constraints).

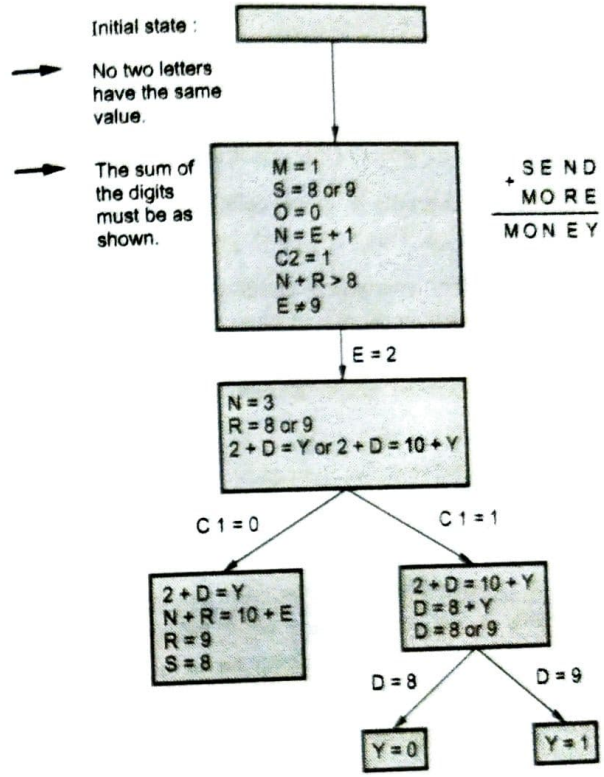


Fig. 3.5.3 The cryptarithmatic problem

3.5.8 General Algorithm for finding Solution in CSP

1. Propagate available constraints -

i) Open all objects that must be assigned values in a complete solution.

ii) Repeat until inconsistency or all objects are assigned valid values : - Select an object and strengthen as much as possible, the set of constraints that apply to object. If set of constraints are different from previous set, then open all objects that share any of these constraints. Remove selected object.

2. If union of constraints discovered above defines a solution then return solution.
3. If union of constraints discovered above defines a contradiction then return failure.

4. Make a guess in order to proceed. Repeat until a solution is found or all possible solutions exhausted;
   i) Select an object with a no assigned value and try to strengthen its constraints.
   ii) Recursively invoke constraint satisfaction with the current set of constraints plus the selected strengthening constraint.

- **CSP Representation as a Constraint Graph**

The CSP can be represented in terms of the constraint graph. A constrain graph has,

1. Nodes as variables (For example-In graph-colouring problem the vertex is variable which needs to be coloured. This vertex will be a node in constraint graph.)
2. Arcs as a constraints (For example - In graph-colouring problem the arc will denote that no two adjacent vertices will have same colour.)

- **Advantages of Constraint Graph**

1. The organization of constraint graph is very much useful for simplification of CSP's solutions.
2. It reduces complexity in exponential manner.

3.5.9 Backtracking Search

- If we apply BFS to generic CSP problems then we can notice that the branching factor at the top level is 'nd', because any of 'd' values can be assigned to any of 'n' variables. At the next level, the branching factor is '(n – 1) d' and so on for n levels. Therfore a tree with 'n! * d^n' leaves is generated even though there are only 'd^n' possbile complete assignments.

3.5.9.1 Basic Steps in Backtracking Search

1. Depth-first search selects values for single variable at a given time.
2. Apply constraint to the variable when variable is selected.
3. Backtracking action-performed if a variable has no legal values left to assign.
4. Backtracking search is basic uninformed algorithm for CSPs.

3.5.9.2 Backtracking Search Algorithm

1. Consider a CSP problem.
2. Apply backtracking search. If backtracking search successful returns a solution else, a failure state which return a procedure recursive-backtracking.
3. Procedure recursive-backtracking starts with empty set and takes input as CSP problem.

   If complete assignement possible or assignment done then return actual assignment.
4. Variable is assigned a specific value.
5. The relative constraint is a set which is taken as input.
6. If value is complete and consistent according to constraints then assign value to variable, add that to list.
7. Call the recursive backtracking until result or failure is reached.
8. Every time recursive-backtracking sustains result (i.e. assignment.)
9. If result is failure then remove variable with specific value from assignment. It will return failure status.

3.5.9.3 Limitations of Backtracking

1. A success function is generated with above algorithm. But backtracking is not effective for very large problems.
2. The general performance is not so good. Domain specific heuristic function combined with uninformed search algorithm can generate better searching result.

Improvement in Backtracking Search

We can also solve CSP without knowing the domain-specific knowledge. Here we need to design a module which resolve following queries : -

1. Which variable is assigned in next step and what order values can be tried ?
2. Impact of current variable assignments on unassigned variables.
3. Avoidance of known failed path in the potential paths.

3.5.10 Commutative Problem

A problem is commutative if the order of application of any given set of actions has no effect on the outcome. This is the case for CSPs because, when assigning values to variables, we reach the same partial assignment, regardless of order. Therefore, all CSP search algorithms generate successors by considering possible assignments for only a single variables at each node in the search tree.

For example -

In graph-colouring problem, if vertex 1 is coloured with red then vertex 2 (adjacent to vertex 1) will have two choices either green or blue but never red.

3.5.11 Selection of next Unassigned Variable

Any general purpose method for solving CSP suffers from making a choice of next unassigned variable.

Variable and Value Ordering : -

- Choosing a variable is critical to performance.
- The efficiency of search algorithms depends considerably on the order in which variables are considered for instantiations.
- This ordering affects the efficiency of the algorithm.
- There exist various heuristics for dynamic or static ordering of values and variables.

Techniques for selecting next best unassigned variable : -

1. Minimum Remaining Values (MRV)

1. It is also called as "Most constrained variable or fail-first heuristic".
2. Backtracking combined with MRV gives better performance.
3. **Rule :** Choose a variable with the fewest legal moves.
4. It answers-Which variable shall we try first ?

2. Degree heuristic

1. It helps to choose next better state.
2. **Rule :** Select variable involved in highest number constraints on other unassigned variable.
3. Degree heuristic is very useful as a tie breaker among MRV variable.
4. It answers-In what order should variable values can be tried ?

3. Least constraining value

1. The least constraining value is also effective method in some application.
2. **Rule :** Choose the least constraining value from many variable i.e. the one that leaves the maximum flexibility for subsequent variable assignment.
3. It can be combined with MRV for fast selection.

3.5.12 Passing Information Through Constraints

While selecting unassigned variable for computation if we look at some of the constraints earlier in the search (or even before search begins), we can drastically reduce search space. **Following are techniques which are helpful for earlier constraints checks.**

3.5.12.1 Forward Checking (FC)

1. Forward checking is one of the potential technique which uses constraints more effectively during search.
2. It removes values in neighbouring unassigned variables domain that conflict with assigned variable.
3. Forward checking uses MRV to select assigned variable.
4. Backtracking search is performed if failure occurs.
5. Search terminates when any variable has no legal values.
6. Generic method,
   i) Assigns variable X.
   ii) Forward checking remarks unassigned variable Y connected to X.
   iii) Remove values from D, where value is inconsistent with X.

3.5.12.2 Constraint Propagation

1. A combined approach of heuristic plus forward checking gives more reliable, accurate and efficient results than a singular approch.
2. The forward checking propagates information from assigned to unassigned variables but can not avoid or detect all failure.
3. Constraint propagation repeatedly enforces constraints locally.
4. The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking. Here "arc" refers to a directed arc in the constraint graph.

Constraint Propagation using Arc Consistency

1. It is fast method of constraint propagation.
2. $X \rightarrow Y$ is consistent if (for every value of X there is some allowed value Y. For example $[V_2 \rightarrow V_1$, is consistent iff V_1 = Red, V_2 = Blue] (i.e. for every value of x in X there is some allowed value y in Y). This is directed property example - $[V_1 = V_2]$

   $V_1 = V_2$ is consistent iff

   V_1 = Red and V_2 = Blue
3. As directed arcs between variables represent the domains of specified variables, they are consistent with each other.
4. Constraint propagation can be applied as preprocessing or propagation step.
   i) Before search-Preprocessing.
   ii) After search-Propagation.

5. The procedure for maintaining arc consistency, can be applied repeatedly.

Constraint Propagation using K-consistency

1. Arc consistency is not capable of detecing all inconsistencies. Partial assignments [V_1 = Red, V_2 = Red] are inconsistent.
2. K-consistency is very strong form of constraint propagation.
3. A CSP is K-consistency if for any set of K-1 variables and for any consistent assignment to those variable a consistent value can always be assigned to any K^{th} variable.

Note -

1. Consistency means that each individual variable by itself is consistent; this is also called as node consistency.
2. Consistency is the same as arc consistency.
3. Consistency means that any pair of adjacent-variables can always be extended to a third neighboring variable; this is also called as path consistency.
4. Strongly, K-consistency graph exhibit some properties mentioned below -

   i) It is K-consistent.

   ii) It is also (K - 1) consistent, (K - 2) consistent all the way down to 1 consistent.
5. This is an idealist solution which requires O (nd) time instead of O (n^2d^3).

An exponential order time is required for establishing n-consistency in the worst case.

3.5.13 Local Search for CSP

1. It is most powerful search for CSP's.
2. Backtrack search require more time in dynamic environment which can be reduced by local search.
3. It uses complete state formulation as follows : -

   a) The initial state which assigns value to every variable.

   b) Successor function alters value of one variable for each instance.

   c) For example - 8-queen problem

Initial state : A random configuration of 8-queens in 8 columns.

Successor function : -

Function 1 : It picks one queen and moved to elsewhere in its columns.

OR

Function 1 : Each column can have queen in a permutations of the 8 rows.

3.5.13.1 MIN-CONFLICTS Algorithm for Solving CSPs by Local Search

1. Min-conflict heuristic select new value that result in a minimum number of conflicts with the other variable.
2. The initial state may be choosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn.
3. The conflicts procedure that counts the number of constraints violated by a particular value, given the rest of the current assignments.

The Algorithm

MIN-CONFLICTS

Inputs : CSP, a constraint satisfaction problem.

Max-steps, the number of steps allowed before giving up.

Output : A solution or failure.

Current-An initial complete assignment for CSP.

for i = 1 to max-steps do

If current is a solution for CSP then return current.

var - a randomly chosen conflicted variable from VARIABLE [CSP].

value - the value v for **var** that minimizes CONFLICTS (var, v, current, csp).

set var = value in current.

return failure.

Advantages of Min-conflict

1. The key feature is that the runtime required for min-conflict is independent of problem size.

   For example - It can solve million - queen problem in an average 50 steps.
2. It also works for hard-problems.
3. Local search is also used in an online setting.

   For Example - in scheduling problem like weekly outline schedule.

- **Example -** of Min conflit

  Consider 4-queen example shown in Fig 3.5.4 in 3 parts (a), (b), (c).

[Figure depicts how 'h' value changes per step]

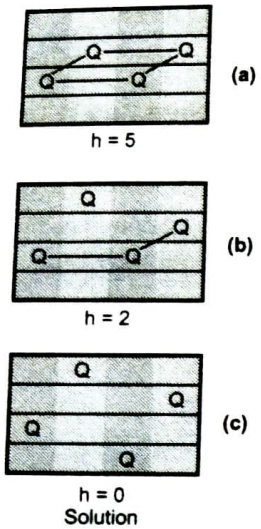


Fig. 3.5.4 Min conflict example

3.5.14 Dealing with Special Constraints

Realistic problem has very special type (real in nature) of constraints which occurs frequently. **Example -**

1. **All-diff constraints :** It enforces the rule that all the variable in state space must have distinct values (as in the cryptorithmetic problem).
2. **Atmost constraints or resource constraints :**

i) These are most important higher order constraints.

ii) It bounds propagation for large value domains which are widely used in practical constraint problem.

3.5.15 Intelligent Backtracking

We observed in forward checking if inconsistency or failure occurs then we need to apply backtracking.

If backtracking is done efficiently then we can reduced total time.

Chronological Backtracking

1. It is one of the standard form (i.e. try different value for preceding variable).
2. In chronological backtracking most latest decision points are always revisited.
3. It has potential to back, up to preceding variable.

4. A more intelligent approach to backtracking is to go all the way back to one of the set of variables that caused the failure. **This set is called the conflict set;** for example, in map colouring problem (discussed in section 3.5.3.1), the conflict set for M is {N, NW, NE}, where N, MW, NE are already coloured regions. In general, the conflict set for variable X is the set of previously assigned variables that are connected to X by constraints.
5. The backjumping method backtracks to the most recent variable in the conflict set.

If no legal value is found, it should return the most recent element of the conflict set along with the failure indicator.

3.5.16 Structure of a Problem (Which can be used for finding quick solution)

A technique to represent a problem in simple way, is to decompose the problem in to many subproblems.

Subproblems can be independent or they can be connected. If the subproblems are totally independant then we have very easy way to solve the problem in totality. We can solve each subproblem independantly and then combine the solutions.

Many practical CSP subproblems are connected. The simple case is when the constraint graph forms a tree. Any two variables are connected by atmost one path.

3.5.17 Algorithm for Solving Tree-Structured CSP in Linear Time

1. Choose any variable as the root of the tree, and order the variables from the root to the leaves in such a way that every node's parent in the tree precedes it in the ordering.
2. Label the variables $X_1, \ldots, X_n$ in order. Now, every variable except the root has exactly one parent variable.

Consider following example,

a) The constraint graph of a tree-structure CSP is,

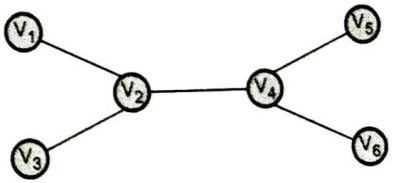


Fig. 3.5.5 Constraint graph

b) A linear ordering of the variables consistent with the tree with V_1 as the root. Refer Fig. 3.5.6.

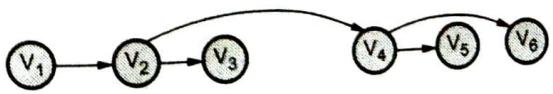


Fig. 3.5.6 Linear ordering of variables

3. For j from n down to 2, apply arc consistency to the arc (X_i, X_j) where X_i is the parent of X_j, removing values from domain $[X_i]$ as necessary.
4. For j from 1 to n, assign any value for X_j consistent with the value assigned far X_i, where X_i, is the parent of X_j.

Note - The complete algorithm runs in time **O (nd^2).**

3.5.18 Graph Structured CSP

If we can reduce graph to a tree then solving graph structured CSP would be better in terms of time.

Reducing a graph to tree can be done in two ways.

1. Remove the nodes. 2. Collapse the nodes.

3.5.18.1 Remove the Nodes Approach (Cutset Conditioning)

1. Choose a subset S from VARIABLES [csp] such that constraint graph becomes a tree after removal of S. **S is called as a cycle cutset.**
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   i) Remove from the domains of the remaining variables any values that are inconsistant with the assignment for S.
   ii) If the remaining CSP has a solution, return it together with the assignment for S.

Note :

1. If the cycle cutset has size C, then the total runtime is O (d^c, $(n - c)\ d^2$).
2. If the graph is "nearly a tree" then c will be small and the savings over straight backtracking will be huge.

For example -

Consider the following graph,

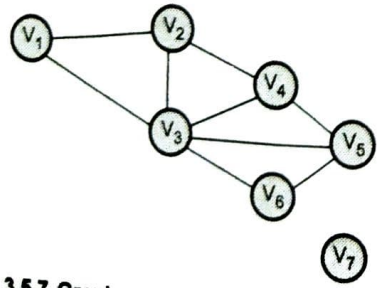


Fig. 3.5.7 Graph used for cutset conditioning

If we remove V_3 then the constraint graph is,

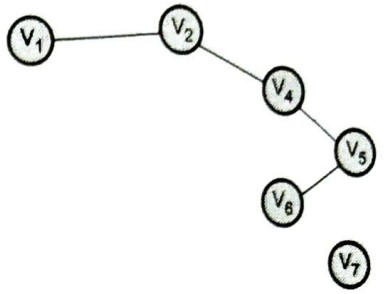


Fig. 3.5.8 Applying cutset on graph

3.5.18.2 Collapse the Node (Tree Decomposition) Approach

1. The approach is based on constructing a tree decomposition of the constraint graph into a set of connected subproblems.
2. Each subproblem is solved independantly and the resulting solutions are then combined.
3. A tree decomposition must satisfy the following three requirements :
   a) Every variable in the original problem appears in at least one of the subproblems.
   b) If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
   c) If a variable appears in two subproblems in the tree, it must appear in every subproblems along the path connecting those subproblems.
4. A given constraint graph admits many tree decompositions. In choosing a decomposition, the aim is to make the subproblems as small as possible.
5. The tree width of a tree decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions.
6. CSPs with constraint graphs of bounded tree width, are solvable in polynomial time. Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

For example -

Consider the following graph -

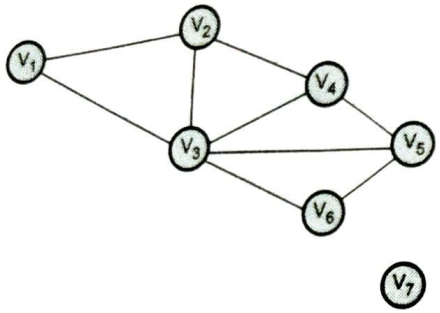


Fig. 3.5.9 Graph used for tree decomposition

A tree decomposition of above graph is,

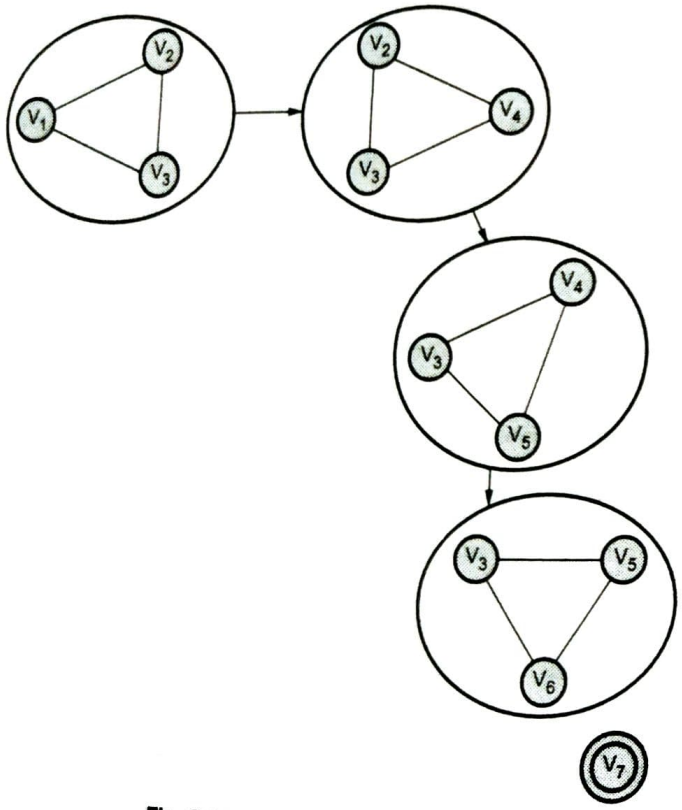


Fig. 3.5.10 Tree decomposition

3.6 Means-Ends Analysis

GTU : Winter-12, 18, Summer-14, 17, 20

- Most of the search strategies either reason forward or backward however, often a mixture of the two directions is appropriate. Such mixed strategy would make it possible to solve the major parts of problem first and solve the smaller problems that arise when combining them together. Such a technique is called "Means - Ends Analysis".
- MEA (Means-Ends Analysis) is a problem solving strategy first introduced in GPS (General Problem Solver) [Newell and Simon, 1963]. The search process over the problem space combines aspects of both forward and backward reasoning in that both the condition and action portions of rules are looked at when considering which rule to apply.
- The means-ends analysis process centers around finding the difference between current state and goal state. The problem space of means - ends analysis has an initial state and one or more goal state, a set of operators with a set of preconditions. Their application and the difference functions computes the difference between two state s(i) and s(j). A problem is solved using means - ends analysis using following steps :
  1. Computing the current state s1 to a goal state s2 and computing their difference D12.
  2. Satisfy the preconditions for some recommended operator OP, which is selected, to reduce the difference D12.
  3. The operator OP is applied if possible. If not, the current state is solved. A goal is created and means-ends analysis is applied recursively to reduce the sub goal.
  4. If the sub goal is solved then the state is restored and work resumed on the original problem.

Means-ends analysis is useful for many human planning activities. Consider the example of planning for an office worker. Suppose we have a different table of three rules :

1. If in our current state we are hungry , and in our goal state we are not hungry , then either the "Visit hotel" or "Visit canteen " operator is recommended.
2. If our current state we do not have money, and if in your goal state we have money, then the "Visit our bank" operator or the "Visit secretary" operator is recommended.
3. If our current state we do not know where something is , need in our goal state we do know, then either the "Visit office enquiry", "Visit secretary" or "Visit co worker " Operator is recommended.

- Differences between the current and goal states are used to propose operators which reduce the differences. The correspondence between operators and differences may be provided as knowledge in the system (in GPS this was known as a Table of Connections) or may be determined through some inspection of the operators if the operator action is penetrable. This later case, which is true of **STRIPS-like** operators, allows task-independent correlation of differences to the operators which reduce them. When knowledge is available concerning the importance of differences, the most important difference is selected first to further improve the average performance of MEA over other brute-force search strategies. However, even without the ordering of differences according to importance, MEA improves over other search heuristics (again in the average case) by focusing the problem solving on the actual differences between the current state and that of the goal.
- In operator sub-goaling, backward chaining is used in which first the operators are selected and then sub goals are set up to establish the preconditions of the operators. If the operator does not produce the goal state, then second sub problem is produced that can reach to goal. If the difference was chosen correctly and if the operator applied is effective at reducing the difference then the two sub problems would be easier to solve than one original problem. The MEA process is then recursively applied.

The MEA algorithm can be summarized as below :

1. Until the goal is reached or no more procedures are available do the steps from 2 to 4.
2. Describe the current state, the goal state and the differences between the two.
3. Use the difference to describe the procedure that would be expected to get nearer to goal.
4. Use the procedure and update current state.
5. If goal is reached then algorithm halts with the success else it halts with failure.

Example 3.6.1 *Consider the following initial and goal configuration for 8-puzzle problem. Draw the search tree. Apply A* algorithm to reach from initial state to goal state and show the solution. Consider Manhattan distance as a heuristic function (i.e. sum of the distance that the tiles are out of place).* **GTU : Summer-17, Marks 7**

Initial state

1	2	3
7	8	4
6		2

Goal state

1	2	3
8		4
7	6	5

Solution : Given :

Initial state

1	2	3
7	8	4
6		5

Goal state

1	2	3
8		4
7	6	5

Hence the goal state (solution)

Applying A* algorithm to reach to goal state,

Heuristic function used : = Manhattan distance
i.e. [Sum of the distance that the tiles are out of place].

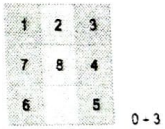
Let : For each node,

$$\hat{f}(n) = \hat{g}(n) + h(n)$$

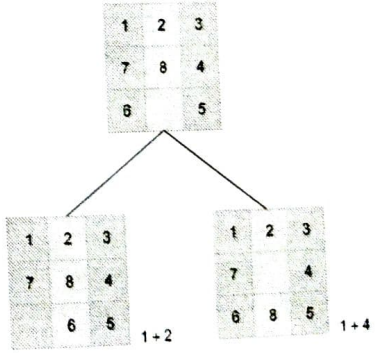
Where,

$\hat{g}(n)$ is an estimate of the 'depth' of 'n' in the graph and h(n) is an heuristic evaluation of node 'n'.

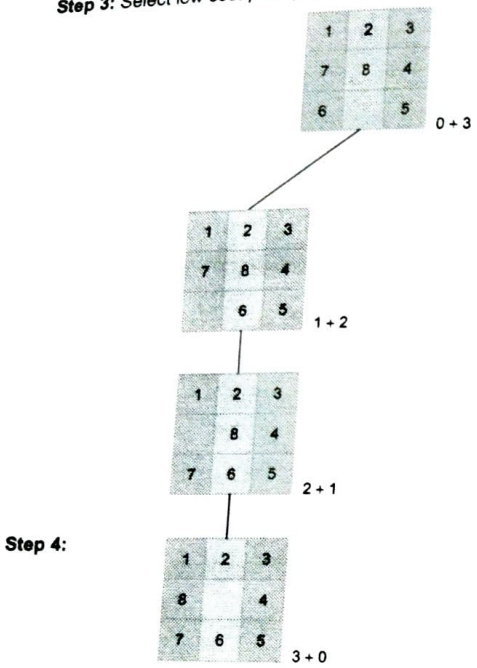
Step 1:



Step 2:



Step 3: Select low cost path (lowest f(n))



Answer in Brief

1. *When is the class of problem said to be intractable ?* **(Refer section 3.1)**
2. *What is the power of heuristic search ? Why does one go for heuristic search ?* **(Refer section 3.1.10)**
3. *What are the advantages of heuristic function ?* **(Refer section 3.1)**
4. *State the reason when hill climbing often get stuck.* **(Refer section 3.2.2)**
5. *When a heuristic function h is said to be admissible ? Give an admissible heuristic function for TSP.* **(Refer section 3.1.10)**
6. *What do you mean by local maxima with respect to search technique ?* **(Refer section 3.2.2)**
7. *What is heuristic search ?* **(Refer section 3.1.10)**
8. *Explain heuristic search with an example.* **(Refer section 3.1.10)**
9. *How to minimize total estimated solution cost using A* search with an example ?* **(Refer section 3.1.6)**
10. *Explain the A* search and give the proof of optimality of A*.* **(Refer section 3.1.6)**
11. *Describe hill climbing, random restart hill climbing and simulated annealing algorithms. Compare their merits and demerits.* **(Refer section 3.2)**

12. *How does hill climbing ensure greedy local search ? What are the problems of hill climbing ?* **(Refer section 3.2)**
13. *How does genetic algorithm come up with optimal solution ?* **(Refer section 3.2.6)**
14. *What is heuristic ? For each of the following type of problem give good heuristics function. i) Block world problem ii) Missionaries and cannibals.* **(Refer section 3.1.10)**
15. *What is heuristics ? A* search uses a combined heuristic to select the best path to follow through the state space toward the goal. Define the two heuristics used (h(n) and g(n)).* **(Refer section 3.1.10)**
16. *Best first search used both OPEN list and a CLOSED list. Describe the purpose of each for the Best-First algorithm. Explain with suitable example.* **(Refer section 3.1)**
17. *Hill climbing is a standard iterotive improvement algorithm similar to greedy best-firsts search. What are the primary problems with hill climbing ?* **(Refer section 3.2)**
18. *What is heuristics ? Explain any heuristics search method. Justify how heuristics function helps in achieving goal state.* **(Refer section 3.1.10)**
19. *What is CSP ?* **(Refer section 3.5)**
20. *Apply constrain satisfaction algorithm to a cryptarithmetic problem given below*

$$\begin{array}{r} SEND \\ +\ MORE \\ \hline MONEY \end{array}$$

(Refer section 3.5)

21. *How CSPs are defined ? Represent map colouring as CSP. Use red, green, blue to colour the map.* **(Refer section 3.5)**
22. *Give the brief summary of backtracking search for CSP.* **(Refer section 3.5)**

3.7 University Questions with Answers

Summer - 12

Q.1 *What is Hill Climbing ? Explain Simple Hill Climbing and Steepest Ascent Hill Climbing.* **(Refer section 3.2.2)** **[7]**

Q.2 *Solve the following cryptarithmetic problem.*

$$\begin{array}{r} S\ E\ N\ D \\ +\ \\ \underline{M\ O\ R\ E} \\ M\ O\ N\ E\ Y \end{array}$$

(Refer section 3.5) **[7]**

Winter - 12

Q.3 *Explain A* algorithm.* **(Refer section 3.1.6)** **[7]**

Q.4 *Explain steepest ascent Hill climbing algorithm.* **(Refer section 3.2.2)** **[7]**