**Velammal College of Engineering and Technology, Madurai**

**Department of Information Technology**

**2016-2017 Even Semester**

**Course Code-Title : CS6659 ARTIFICIAL INTELLIGENCE Degree/Year/Semester: B.TECH-IT/**III/VI

**Name of the Instructor: Ms. S. Nirmala**

**UNIT I**

**INTRODUCTION TO Al AND PRODUCTION SYSTEMS**

**Syllabus:** Introduction to AI-Problem formulation, Problem Definition -Production systems, Control strategies, Search strategies. Problem characteristics, Production system characteristics -Specialized production system- Problem solving methods - Problem graphs, Matching, Indexing and Heuristic functions -Hill Climbing-Depth first and Breath first, Constraints satisfaction - Related algorithms, Measure of performance and analysis of search algorithms.

**Introduction to AI**

**Definition**

1. Intelligence exhibited by machines or software.
2. A field of study which studies how to create computers and computer software that are capable of intelligent behavior
3. “The study and design of intelligent agents”, in which an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success
4. “The science and engineering of making intelligent machines"

**Application:**

* Autonomous planning and scheduling
* Game playing
* Autonomous control
* Diagnostics
* Logistics planning
* Robotics
* Voice Recognition

**Problem Characteristics**

* Decomposable?
* Can partial steps be ignored or undone ?
* Predictable ?
* Is “good” solution easily recognizable ?
* Is Knowledge Base consistent ?
* How much Knowledge is needed ?
* Stand–alone vs. Inter–active.

**Problem Solving Methodology:**

The followings are the things to build a system to solve a particular problem

1. Define the problem precisely
2. Analyze the problem**(SEARCH)**
3. Knowledge representation
4. Choose the best problem solving techniques

**Definition of the problem:**

The solution of many problems can be described by finding a sequence of actions that lead to a desirable goal. Each action changes the state and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

A well-defined problem can be described by:

* **Initial state** :Any Configuration
* **Operator or successor function** - A successor function is needed to move between different states. Representation of the successor function:
	+ Conditions of applicability
	+ Transformation function
* **State space** - all states reachable from initial by any sequence of actions
* **Path** - sequence through state space
* **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
* **Goal test** - test to determine if at goal state

**Example**

**The 8-puzzle**

An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach the goal state, as shown in the following figure

**Example: The 8-puzzle**

|  |
| --- |
|  |

The problem formulation is as follows :

* **State space** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
* **Initial state** : Any state can be designated as the initial state.
* **Successor function** :”Blank Move”
	+ Condition: the move is within the board
	+ Transformation: blank moves *Left*, *Right*, *Up*, or *Down*
* **Goal Test** : This checks whether the state matches the goal state
* **Path cost** : Each step costs 1,so the path cost is the number of steps in the path.

**The 8**-**puzzle** belongs to the family **of sliding-block puzzles**, which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete.

1. The **8**-**puzzle** has 9!/2 = 181,440 reachable states and is easily solved.
2. The **15 puzzle** ( 4 x 4 board ) has around 1.3 trillion states ,an the random instances can be solved optimally in few milli seconds by the best search algorithms.
3. The **24-puzzle** (on a 5 x 5 board) has around 1025 states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

# Search in Artificial Intelligence

Search plays a major role in solving many Artificial Intelligence (AI) problems. Search is a universal problem-solving mechanism in AI. In many problems, sequence of steps required to solve is not known in advance but must be determined by systematic trial-and-error exploration of alternatives.

The problems that are addressed by AI search algorithms fall into three general classes:

* Single-agent path-finding problems,
* Two-players games,
* Constraint-satisfaction problems

# Single-agent path-finding problems

* **Classic examples**:
1. path-finding problems
2. Puzzles
* **Real-world problems**
1. Traveling salesman problem,
2. Vehicle navigation,
3. Wiring of VLSI circuits.

In each case, the task is to find a sequence of operations that map an initial state to a goal state.

# Two-players games

Two-player games are two-player perfect information games. Chess, checkers.

# Constraint Satisfaction Problems

Eight Queens Problem is the best example. The task is to place eight queens on an 8\*8 chessboard such that no two queens are on the same row, column or diagonal. Real-world examples of constraint satisfaction problems are planning and scheduling applications.

# Problem Space

Problem space is a set of states and the connections between to represent the problem. Problem space graph is used to represent a problem space. In the graph, the states are represented by nodes of the graph, and the operators by edges between nodes. Although most problem spaces correspond to graphs with more than one path between a pair of nodes, for simplicity they are often represented as trees, where the initial state is the root of the tree. The cost of the simplification is that any state that can be reached by two different paths will be represented by duplicate nodes thereby increasing the tree size. The benefit of using tree is that the absence of cycles greatly simplifies many search algorithms. One feature that distinguishes AI search algorithms from other graph-searching algorithms is the size of the graph involved.

For example, the entire chess graph is estimated to contain over 10^40 nodes. Even a simple problem like twenty-four puzzle contains almost 10^25 nodes.

As a result, the problem-space graphs of AI problems are never represented explicitly by listing each state, but rather are implicitly represented by specifying an initial state and a set of operators to generate new states from existing states.

Moreover the size of an AI problem is rarely expressed as the number of nodes in its problem-space graph. The two parameters of a search tree that determine the efficiency of various search algorithms are its branching factor and its solution depth.

The branching factor is nothing but average number of children of a given node. For example in eight-puzzle problem, the average branching factor is square-root (3) or about 1.732.

The solution depth of a problem instance is the length of a shortest path from the initial state to a goal state or the length of a shortest sequence of operators that solve the problem.

**Production systems:**

 Production system can be defined as set of rules that have been written by the programmer that define how to solve the particular problems *or* it can be defined as the process of solving the problem can usefully be modeled as a production system

**Characteristics:**

 In order to solve a problem: We must first reduce it to one for which a precise statement can be given. This can be done by defining the problem’s state space (start and goal states) and a set of operators for moving that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modeled as a production system.

 How to decide which rule to apply next during the process of searching for a solution to a problem? The two requirements of good control strategy are that it should cause motion. It should be systematic

***Elements of Production Systems:***

1. Uninformed Search Algorithms *or* Control Strategies

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

* Breadth First Search
* Depth First Search
1. Informed Search *or* Heuristics Search Algorithm

 **Strategies** that know whether one non goal state is “more promising” than another are called **Informed search or heuristic search** strategies.

* Generate and Test
* Hill climbing
* Best First Search
* A\*
* Problem Reduction
* Constraint satisfaction
* Means-ends analysis

**MEASURING PROBLEM-SOLVING PERFORMANCE**

The output of problem-solving algorithm is either failure or a solution.

(Some algorithms might struck in an infinite loop and never return an output)

The algorithm’s performance can be measured in four ways:

* **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
* **Optimality** : Does the strategy find the optimal solution
* **Time complexity:** How long does it take to find a solution?
* **Space complexity:** How much memory is needed to perform the search?

**Breadth First Search** (BFS)

**Breadth First Search** (BFS) searches breadth-wise in the problem space. Breadth-First search is like traversing a tree where each node is a state which may a be a potential candidate for solution. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

**Algorithm: Breadth-First Search**

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
	* 1. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
		2. For each way that each rule can match the state described in E do:

i) Apply the rule to generate a new state.
ii) If the new state is the goal state, quit and return this state.
iii) Otherwise add this state to the end of NODE-LIST

Since it never generates a node in the tree until all the nodes at shallower levels have been generated, *breadth-first search* always finds a shortest path to a goal. Since each node can be generated in constant time, the amount of time used by Breadth first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution d. Since the number of nodes at level d is bd, the total number of nodes generated in the worst case is b + b2 + b3 +… + bd

 i.e. O(bd) , the asymptotic time complexity of breadth first search.

**Example**:



In this tree with nodes starting from root node, R at the first level, A and B at the second level and C, D, E and F at the third level. If we want to search for node E then BFS will search level by level. First it will check if E exists at the root. Then it will check nodes at the second level. Finally it will find E a the third level.

**Advantages of Breadth-First Search**

1. Breadth first search will never get trapped exploring the useless path forever.
2. If there is a solution, BFS will definitely find it out.
3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

**Disadvantages of Breadth-First Search**

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is O(bd). As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.
2. If the solution is farther away from the root, breath first search will consume lot of time.

**Depth First Search (DFS)**

Depth First Search (DFS) searches deeper into the problem space. Breadth-first search always generates successor of the deepest unexpanded node. It uses last-in first-out stack for keeping the unexpanded nodes. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack.

**Algorithm: Depth First Search**

1. If the initial state is a goal state, quit and return success.
2. Otherwise, loop until success or failure is signaled.
a) Generate a state, say E, and let it be the successor of the initial state. If there is no successor, signal failure.
b) Call Depth-First Search with E as the initial state.
c) If success is returned, signal success. Otherwise continue in this loop.

**Advantages of Depth-First Search**

* The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
* The time complexity of a depth-first Search to depth d is O(b^d) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.
* If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

**Disadvantages of Depth-First Search**

* The disadvantage of Depth-First Search is that there is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree. One solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.
* Depth-First Search is not guaranteed to find the solution.
* And there is no guarantee to find a minimal solution, if more than one solution exists.

**Example**:

 The following figure illustrates the 8-puzzle searched by a production system with loop detection and depth-bound



**INFORMED SEARCH OR HEURISTIC SEARCH** **STRATEGIES**.

In order to solve larger problems, domain-specific knowledge must be added to improve search efficiency. Information about the problem includes the nature of states, cost of transforming from one state to another, and characteristics of the goals. This information can often be expressed in the form of heuristic evaluation function, say f (n,g), a function of the nodes n and/or the goals g.

**Generate-And-Test Algorithm**

Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution.

**Algorithm: Generate-And-Test**

* 1. Generate a possible solution.
	2. Test to see if this is the expected solution.
	3. If the solution has been found quit else go to step 1.

 Potential solutions that need to be generated vary depending on the kinds of problems. For some problems the possible solutions may be particular points in the problem space and for some problems, paths from the start state.



 Generate-and-test, like depth-first search, requires that complete solutions be generated for testing. In its most systematic form, it is only an exhaustive search of the problem space. Solutions can also be generated randomly but solution is not guaranteed. This approach is what is known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

**Systematic Generate-And-Test**

 While generating complete solutions and generating random solutions are the two extremes there exists another approach that lies in between. The approach is that the search process proceeds systematically but some paths that unlikely to lead the solution are not considered. This evaluation is performed by a heuristic function.

 Depth-first search tree with backtracking can be used to implement systematic generate-and-test procedure. As per this procedure, if some intermediate states are likely to appear often in the tree, it would be better to modify that procedure to traverse a graph rather than a tree.

**Generate-And-Test and Planning**

 Exhaustive generate-and-test is very useful for simple problems. But for complex problems even heuristic generate-and-test is not very effective technique. But this may be made effective by combining with other techniques in such a way that the space in which to search is restricted.

 An AI program DENDRAL, for example, uses plan-Generate-and-test technique. First, the planning process uses constraint-satisfaction techniques and creates lists of recommended and contraindicated substructures. Then the generate-and-test procedure uses the lists generated and required to explore only a limited set of structures.

 Constrained in this way, generate-and-test proved highly effective. A major weakness of planning is that it often produces inaccurate solutions as there is no feedback from the world. But if it is used to produce only pieces of solutions then lack of detailed accuracy becomes unimportant.

**Hill climbing**

* Heuristic function to estimate how close a given state is to a goal state.
* Generate-and-test + direction to move.
* Searching for a goal state = Climbing to the top of a hill
* **Simple Hill Climbing**

Evaluation function as a way to inject task-specific knowledge into the control process

**Algorithm**

1. Evaluate the initial state.
2. Loop until a solution is found or there are no new operators left to be applied:

 - Select and apply a new operator

 - Evaluate the new state:

 goal → quit

 better than current state → new current state

**Example: coloured blocks**

 Heuristic function: the sum of the number of different colours on each of the four sides (solution = 16).

* **Steepest-Ascent Hill Climbing (Gradient Search)**
1. Considers all the moves from the current state.
2. Selects the best one as the next state.

Algorithm

1. Evaluate the initial state.
2. Loop until a solution is found or a complete iteration produces no change to current state:

 - SUCC = a state such that any possible successor of the current state will be better than SUCC (the worst state).

 - For each operator that applies to the current state, evaluate the new state:

 goal → quit

 better than SUCC → set SUCC to this state

 - SUCC is better than the current state → set the current state to SUCC.

**Hill Climbing: Disadvantages**

* *Local maximum*

 A state that is better than all of its neighbours, but not better than some other states far away.

* *Plateau*

 A flat area of the search space in which all neighbouring States have the same value.

* *Ridge*

 The orientation of the high region, compared to the set of available moves, makes it impossible to climb up. However, two moves executed serially may increase the height.

* *Ways Out*
* Backtrack to some earlier node and try going in a different direction.
* Make a big jump to try to get in a new section.
* Moving in several directions at once.
* *Hill climbing is a local method:*

 Decides what to do next by looking only at the “immediate” consequences of its choices. Global information might be encoded in heuristic functions.

**Start Goal**

**Local heuristic:**

* +1 for each block that is resting on the thing it is supposed to be resting on.
* -1 for each block that is resting on a wrong thing.

**Start =0 Goal=4**

Step 2:

 **0** **2**



**Global heuristic:**

* For each block that has the correct support structure: +1 to every block in the support structure.
* For each block that has a wrong support structure: -1 to every block in the support structure.

**Start =-6 Goal=+6**



* **Simulated Annealing**

 A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made. To do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state. Lowering the chances of getting caught at a local maximum, or plateau, or a ridge.

**Physical Annealing**

* Physical substances are melted and then gradually cooled until some solid state is reached.
* The goal is to produce a minimal-energy state.
* Annealing schedule: if the temperature is lowered sufficiently slowly, then the goal will be attained.
* Nevertheless, there is some probability for a transition to a higher energy state: e-ΔE/kT.

**Algorithm**

1. Evaluate the initial state.
2. Loop until a solution is found or there are no new operators left to be applied:

 - Set T according to an annealing schedule

 - Selects and applies a new operator

 - Evaluate the new state:

 goal → quit

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

 ΔE **<** 0 → new current state

 else → new current state with probability e-ΔE/kT.

**BEST FIRST SEARCH**

* Depth-first search: not all competing branches having to be expanded.
* Breadth-first search: not getting trapped on dead-end paths.

 Combining the two is to follow a single path at a time, but switch paths whenever some competing path look more promising than the current one.

* OPEN: nodes that have been generated, but have not examined.

 This is organized as a priority queue.

* CLOSED: nodes that have already been examined.

 Whenever a new node is generated, check whether it has been generated before.

Algorithm

1. OPEN = {initial state}.
2. Loop until a goal is found or there are no nodes left in OPEN:

 - Pick the best node in OPEN

 - Generate its successors

 - For each successor:

* new → evaluate it, add it to OPEN, record its parent
* generated before → change parent, update successors

**Example**:



* Greedy search:

h(n) = estimated cost of the cheapest path from node n to a goal state. /\***Neither optimal nor complete**\*\

* Uniform-cost search:

g(n) = cost of the cheapest path from the initial state to node n.

/\***Optimal and complete, but very inefficient** \*\

 **A\* Algorithm**

A\* (A star) is the most widely known form of Best-First search. It evaluates nodes by combining g(n) and h(n)

 f(n) = g(n) + h(n)

Where

g(n) = cost so far to reach n

h(n) = estimated cost to goal from n

f(n) = estimated total cost of path through n

* When h(n) = actual cost to goal
	+ Only nodes in the correct path are expanded
	+ Optimal solution is found
* When h(n) < actual cost to goal
	+ Additional nodes are expanded
	+ Optimal solution is found
* When h(n) > actual cost to goal
	+ Optimal solution can be overlooked
* A\* is optimal if it uses an admissible heuristic
	+ h(n) <= h\*(n) the true cost from node n
	+ if h(n) never overestimates the cost to reach the goal
* Example
	+ hSLD never overestimates the actual road distance

**PROBLEM REDUCTION**

Problem reduction search can be planning how best to solve a problem that can be recursively decomposed into sub problems in multiple ways.

1) Matrix multiplication problem

2) Tower of Hanoi

3) Blocks world problems

4) Theorem proving

**AND-OR Graph**



**AO\* Algorithm:**

1. Let G consists only to the node representing the initial state call this node INTT. Compute h’ (INIT).

2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(I)Trace the marked arcs from INIT and select an unbounded node NODE.

(II)Generate the successors of NODE. If there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not also an ancestor of NODE do the following

(a) Add SUCCESSOR to graph G

(b) If successor is not a terminal node, mark it solved and assign zero to its h ' value.

(c) If successor is not a terminal node, compute it h' value.

(III) Propagate the newly discovered information up the graph by doing the following. Let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;

(a) Select a node from S call if CURRENT and remove it from S.

(b) Compute h' of each of the arcs emerging from CURRENT, Assign minimum h' to CURRENT.

(c) Mark the minimum cost path a s the best out of CURRENT.

(d) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labeled SOLVED.

(e) If CURRENT has been marked SOLVED or its h' has just changed, its new status must be propagate backwards up the graph. Hence all the ancestors of CURRENT are added to S.



**AO\* Search Procedure:**

1. Place the start node on open.

2. Using the search tree, compute the most promising solution tree TP .

3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.

4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.

5. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure.Remove all nodes from open ,with unsolvable ancestors.

6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.

7. Go back to step (2)

 *Note: AO\* will always find minimum cost solution.*

**

**Properties of AO\*:**

• AO\* is a generalization of A\* for AND-OR graphs

• AO\*, like A\*, is admissible if the heuristic function is admissible and the usual assumptions (finite branching factor etc) hold

• AO\*, like A\* is also optimal among the class of heuristic search algorithms that use an additive cost / evaluation function

.

**CONSTRAINT SATISFACTION**

* Many AI problems can be viewed as problems of constrained satisfaction in which the goal is to solve some problem state that satisfies a given set of constraints. Such problems do not require a new search methods. They can be solved using any of the search strategies which can be augmented with the list of constraints that change as parts of the problem are solved. Example of such a problem are
	+ **Crypt-Arithmetic** puzzles.
	+ Many design tasks can also be viewed as constrained satisfaction problems.
	+ N-Queen: Given the condition that no two queens on the same row/column/diagonal attack each other.
	+ Map coloring: Given a map, color three regions in blue, red and black, such that no two neighboring regions have the same color.

**Algorithm:**

1. Until a complete solution is found or all paths have lead to dead ends {
	* Select an unexpanded node of the search graph.
	* Apply the constraint inference rules to the selected node to generate all possible new constraints.
	* If the set of constraints contain a contradiction, then report that this path is a dead end.
	* If the set of constraint describes a complete solution, then report success.
	* If neither a contradiction nor a complete solution has been found, then
		+ Apply the problem space rules to generate new partial solutions that are consistent with the current set of constraints.
		+ Insert these partial solutions into the search graph.

}

Example Problem:

* + Solve the following puzzle by assigning numeral (0-9) in such a way that each letter is assigned unique digit which satisfy the following addition.
	+ Constraints : No two letters have the same value. (The constraints of arithmetic).

* + Initial Problem State

 S = ? ; E = ? ;N = ? ; D = ? ; M = ? ;O = ? ; R = ? ;Y = ?

Carries :

C4 = ? ; C3 = ? ; C2 = ? ; C1 = ?

Constraint equations:

 Y = D + E  C1

 E = N + R + C1 C2

 N = E + O + C2  C3

 O = S + M + C3 C4

 M = C4

* We can easily see that M has to be non zero digit, so the value of C4 =1
1. M = C4 ⇒ **M = 1**
2. O = S + M + C3 🡪 C4

 For C4 =1, S + M + C3 > 9 ⇒

 S + 1 + C3 > 9 ⇒ S+C3 > 8.

 If C3 = 0, then S = 9 else if C3 = 1,

 then S = 8 or 9.

* We see that for S = 9
	+ C3 = 0 or 1
	+ It can be easily seen that C3 = 1 is not possible as O = S + M + C3 ⇒ O = 11 ⇒ O has to be assigned digit 1 but 1 is already assigned to M, so not possible.
	+ Therefore, only choice for C3 = 0, and thus O = 10. This implies that O is assigned 0 (zero) digit.
* **Therefore, O = 0**

 **M = 1, O = 0**

1. Since C3 = 0; N = E + O + C2 produces no carry.
* As O = 0, N = E + C2 .
* Since N ≠ E, therefore, C2 = 1.

Hence N = E + 1

* Now E can take value from 2 to 8 {0,1,9 already assigned so far }
	+ If E = 2, then N = 3.
	+ Since C2 = 1, from E = N + R + C1 , we get 12 = N + R + C1
		- If C1 = 0 then R = 9, which is not possible as we are on the path with S = 9
		- If C1 = 1 then R = 8, then
			* + From Y = D + E , we get 10 + Y= D + 2 .
				+ For no value of D, we can get Y.
	+ Try similarly for E = 3, 4. We fail in each case.
* If E = 5, then N = 6
	+ Since C2 = 1, from E = N + R + C1 , we get 15 = N + R + C1
	+ If C1 = 0 then R = 9, which is not possible as we are on the path with S = 9.
	+ If C1 = 1 then R = 8, then
		- From Y = D + E , we get 10 + Y= D + 5 i.e., 5 + Y = D.
		- If Y = 2 then D = 7. These values are possible.
			* Hence we get the final solution as given below and on backtracking, we may find more solutions.

|  |
| --- |
|  **S = 9 ; E = 5 ; N = 6 ; D = 7 ;**  |
|  **M = 1 ; O = 0 ; R = 8 ;Y = 2**  |



**MEANS-ENDS ANALYSIS**

1. Involves detection of difference between current state and goal state
2. Once difference identified, an operator to reduce the difference must be found
3. But perhaps operator cannot be applied to current state
4. Sub problem of getting to state where operator can be applied
5. Operator may not result in goal state
6. Second sub problem of getting from new state to goal state
7. MEA process applied recursively
8. Each rule (operator) has LHS preconditions and RHS aspects of problem state changed. Difference table of rules and differences they can reduce
9. Problem for household robot: moving desk with 2 things on it from one room to another. Main difference between start and goal state is location. Choose PUSH and CARRY

**EXAMPLE**: Move desk with 2 things on it to new room

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Push** | **Carry** | **Walk** | **Pickup** | **Putdown** | **Place** |
| **Move object** | **\*** | **\*** |  |  |  |  |
| **Move robot** |  |  | **\*** |  |  |  |
| **Clear object** |  |  |  | **\*** |  |  |
| **Get object on object** |  |  |  |  |  | **\*** |
| **Get arm empty** |  |  |  |  | **\*** | **\*** |
| **Be holding object** |  |  |  | **\*** |  |  |

CARRY: preconditions cannot be met

PUSH: 4 preconditions

WALK to object, clear desk using PICKUP and PLACE. After PUSH objects not on desk. Must WALK to collect them and put on table using PICKUP and CARRY

|  |  |  |
| --- | --- | --- |
| **Operator** | **Preconditions** | **Results** |
| **PUSH (obj, loc)** | at(robot,obj)&large (obj) &clear (obj) &arm empty | at(obj, loc) & at (robot, loc) |
| **CARRY (obj, loc)** | at(robot, obj) &Small (obj) | at(obj, loc) &at(robot, loc) |
| **WALK(loc)** | none | At(robot, loc) |
| **PICKUP(obj)** | At(robot, obj) | Holding(obj) |
| **PUTDOWN(obj)** | Holding(obj) | Not holding (obj) |
| **PLACE(obj1, obj2)** | At(robot,obj2) & holding (obj1) | on(obj1, obj2) |

1. Compare CURRENT to GOAL. If no differences, return.

2. Otherwise select most important difference and reduce it by doing the following until success or failure is indicated.

1. Select an as yet untried operator *O* that is applicable to the current difference. If there are no such operators then signal failure.
2. Attempt to apply *O* to the current state. Generate descriptions of two states *O*-START a state in which *O’*s preconditions are satisfied and *O-*RESULT, the state that would result if O were applied in *O-*START.
3. If (FIRST-PART MEA (CURRENT,*O*-START) AND (LAST-PART MEA (*O*-RESULT, GOAL) are successful then signal success.