MODULE 2

****

SYLLABUS

Module 2 (14 hours)

Search Methods: data driven and goal driven search. Depth first and breadth first search, DFS with iterative deepening. Heuristic search-best first search, A * algorithm. AO* algorithm, Constraint Satisfaction. Crypt Arithmetic Problems

****

The State Space Representation of Problems

In the *state space representation* of a problem, the nodes of a graph correspond to partial problem solution *states* and the arcs correspond to steps in a problem-solving process. One or more *initial states*, corresponding to the given information in a problem instance, form the root of the graph. The graph also defines one or more *goal* conditions, which are solutions to a problem instance. *State space search* characterizes problem solving as the process of finding a *solution path* from the start state to a goal.

Arcs of the state space correspond to steps in a solution process and paths through the space represent solutions in various stages of completion. Paths are searched, beginning at the start state and continuing through the graph, until either the goal description is satisfied or they are abandoned. The actual generation of new states along the path is done by applying operators, such as "legal moves" in a game or inference rules in a logic problem or expert system, to existing states on a path. The task of a search algorithm is to find a solution path through such a problem space.

Search algorithms must keep track of the paths from a start to a goal node, because these paths contain the series of operations that lead to the problem solution.

We now formally define the state space representation of problems:

A *state space* is represented by a four-tuple **[N,A,S,GD]**, where:

- **N** is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.
- A is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process.
- S, a nonempty subset of N, contains the start state(s) of the problem.
- GD, a nonempty subset of N, contains the goal state(s) of the problem. The states in GD are described using either:
  - A measurable property of the states encountered in the search.
  - A measurable property of the path developed in the search, for example, the sum of the transition costs for the arcs of the path.

A *solution path* is a path through this graph from a node in S to a node in GD.

EXAMPLE 1 TIC-TAC-TOE

The state space representation of tic-tac-toe appears in following figure.

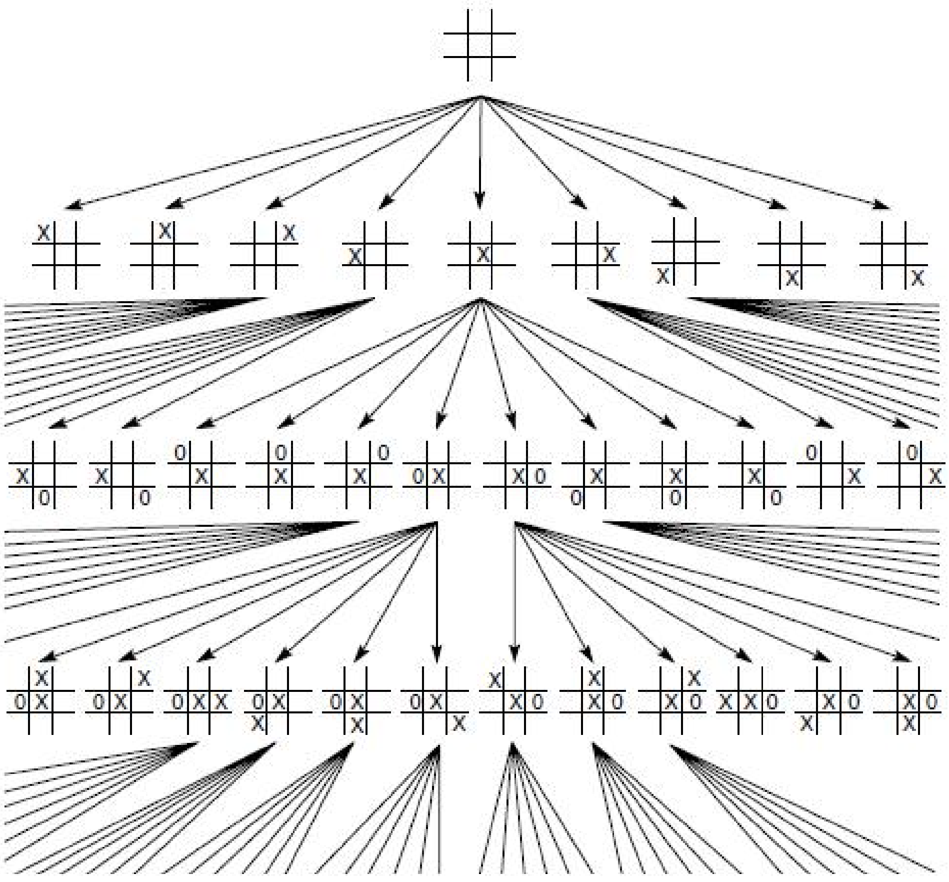


Figure II.5 Portion of the state space for tic-tac-toe.

The start state is an empty board, and the termination or goal description is a board state having three Xs in a row, column, or diagonal (assuming that the goal is a win for X). The path from the start state to a goal state gives the series of moves in a winning game.

The states in the space are all the different configurations of Xs and Os that the game can have. Of course, although there are 39 ways to arrange {blank, X, O} in nine spaces, most of them would never occur in an actual game. Arcs are generated by legal moves of the game, alternating between placing an X and an O in an unused location. The state space is a graph rather than a tree, as some states on the third and deeper levels can be reached by different paths. However, there are no cycles in the state space, because the directed arcs of the graph do not allow a move to be undone. It is impossible to "go back up" the structure once a state has been reached. No checking for cycles in path generation is necessary. A graph structure with this property is called a *directed acyclic graph*, or *DAG*, and is common in state space search and in graphical models.

The state space representation provides a means of determining the complexity of the problem. In tic-tac-toe, there are nine first moves with eight possible responses to each of them, followed by seven possible responses to each of these, and so on. It follows that 9 × 8 × 7 × ... or 9! different paths can be generated. Although it is not impossible for a computer to search this number of paths (362,880) exhaustively, many important problems also exhibit factorial or exponential complexity, although on a much larger scale.

Chess has 10120 possible game paths; checkers has 1040, some of which may never occur in an actual game. These spaces are difficult or impossible to search exhaustively. Strategies for searching such large spaces often rely on heuristics to reduce the complexity of the Search

Example 2 THE 8-PUZZLE

In the *15-puzzle of* differently numbered tiles are fitted into 16 spaces on a grid. One space is left blank so that tiles can be moved around to form different patterns. The goal is to find a series of moves of tiles into the blank space that places the board in a goal configuration. This is a common game that most of us played as children.

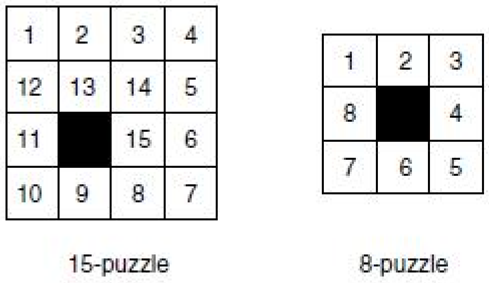


Figure 3.7 The 15-puzzle and the 8-puzzle.

The *8-puzzle* is a 3 × 3 version of the 15-puzzle in which eight tiles can be moved around in nine spaces. Although in the physical puzzle moves are made by moving tiles ("move the 7 tile right, provided the blank is to the right of the tile" or "move the 3 tile down"), it is much simpler to think in terms of "moving the blank space". This simplifies the definition of move rules because there are eight tiles but only a single blank. In order to apply a move, we must make sure that it does not move the blank off the board. Therefore, all four moves are not applicable at all times; for example, when the blank is in one of the corners only two moves are possible.

The legal moves are:

move the blank up	↑
move the blank right	→
move the blank down	↓
move the blank left	←

If we specify a beginning state and a goal state for the 8-puzzle, it is possible to give a state space accounting of the problem-solving process. States could be represented using a simple 3x3 array.

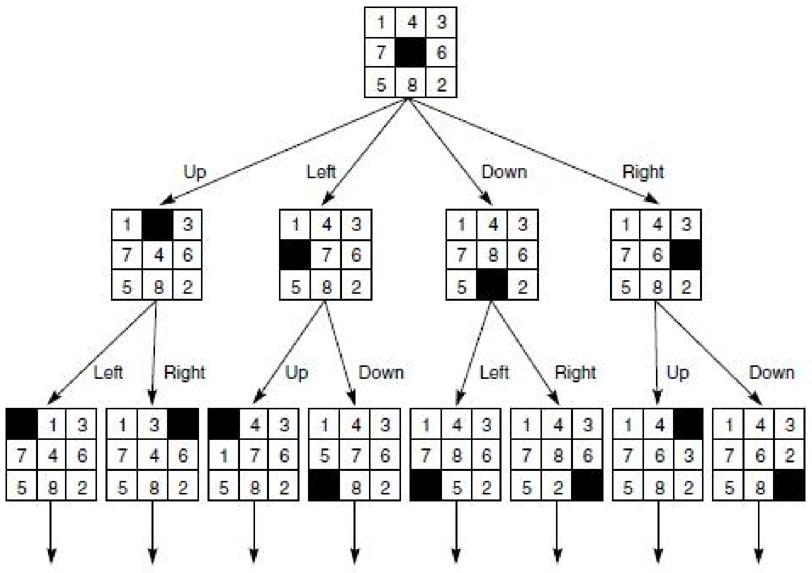


Figure 3.8 State space of the 8-puzzle generated by move blank operations.

Strategies for State Space Search

In *data-driven search*, sometimes called *forward chaining*, the problem solver begins with the given facts of the problem and a set of legal moves or rules for changing state. Search proceeds by applying rules to facts to produce new facts, which are in turn used by the rules to generate more new facts. This process continues until (we hope!) it generates a path that satisfies the goal condition.

An alternative approach is possible: take the goal that we want to solve. See what rules or legal moves could be used to generate this goal and determine what conditions must be true to use them. These conditions become the new goals, or *subgoals*, for the search. Search continues, working backward through successive subgoals until (we hope!) it works back to the facts of the problem. This finds the chain of moves or rules leading from data to a goal, although it does so in backward order. This approach is called *goal-driven* reasoning, or *backward chaining*, and it recalls the simple childhood trick of trying to solve a maze by working back from the finish to the start.

To summarize: data-driven reasoning takes the facts of the problem and applies the rules or legal moves to produce new facts that lead to a goal; goal-driven reasoning focuses on the goal, finds the rules that could produce the goal, and chains backward through successive rules and subgoals to the given facts of the problem.

Data driven and Goal driven

Goal driven search is suggested if :

1. A goal is given in the problem statement or can easily be formulated. In a mathematics theorem prover, for example, the goal is the theorem to be proved. Many diagnostics system consider potential diagnoses in a systematic fashion.
2. There are large no. of rules that match the facts of the problem and thus produce an increase in no of conclusions or goals.In a mathematics theorem

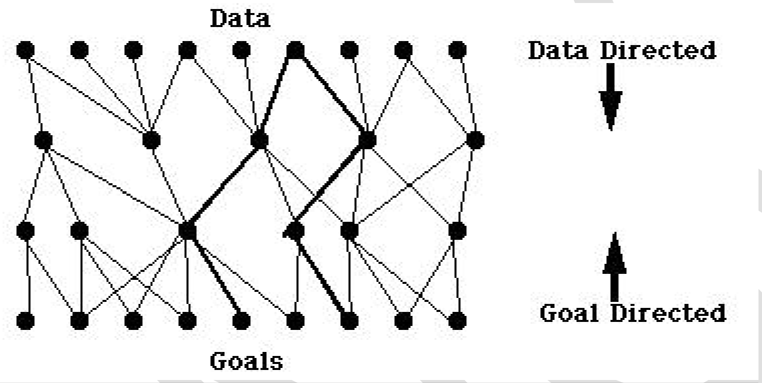
prover, for example, the total no of rules that may be applied to the entire set of axioms.

3. Problem data are not given and must be acquired by the problem solver.In a medical diagnosis program,for example, a wide range of diagnostic tests can be applied.Doctors order only those that are necessary to confirm or deny a particular hypothesis.

Data driven search

1. All or most of the data are given in the initial problem statement.Interpretation problems often fit this mold by presenting a collection of data and asking the system for interpretation.
2. There are large no. of potential goals,but there are only a few ways to use the facts and given information of a particular problem instance.
3. It is difficult to form a goal or hypothesis.

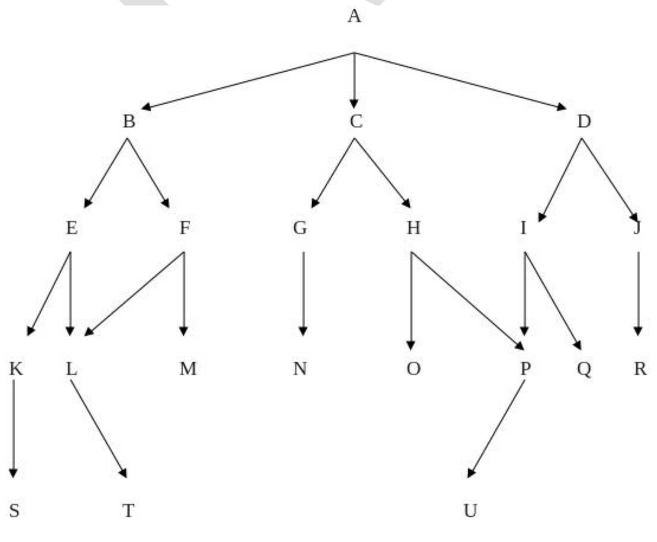
Data driven search uses the knowledge and constraints found in the given data to guide search along lines known to be true



Searching strategies

1. Breadth first search

Consider the graph shown below.



States are labeled (A, B, C. . . .).

CWCL

Breadth first search explores the space in a level by level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next level. A breadth first search of the above graph considers the states in the order

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.

We implement breadth first search using lists, open and closed, to keep track of progress through the state space.

'open' lists states that have been generated but whose children have not been examined. The order in which states are removed from open determines the order of the search.

'closed' records states that have already been examined.

```
void breadth_ f i r s t _ search ( )
{
   open = [ s t a r t ] ;
   closed = [ ] ;
   while ( open not empty )
   {
      Remove the leftmost state from open , call it X;
      if X is a goal ,
      Then
         return SUCCESS;
      else
      {
         Generate children of X;
         Put X on closed ;
         Discard children of X, if already on open or closed ;
         Put remaining children on right end of open ;
      }
   }
   return FAIL ;
}
```

Child states are generated by inference rules, legal moves of a game or other state transition operators. Each iteration produces all children of the state X and adds them to 'open'.

Note that 'open' is maintained as a queue (FIFO) data structure. States are added to the right of the list and removed from the left.

A trace of the breadth first search on the graph appears below.

SNGCE

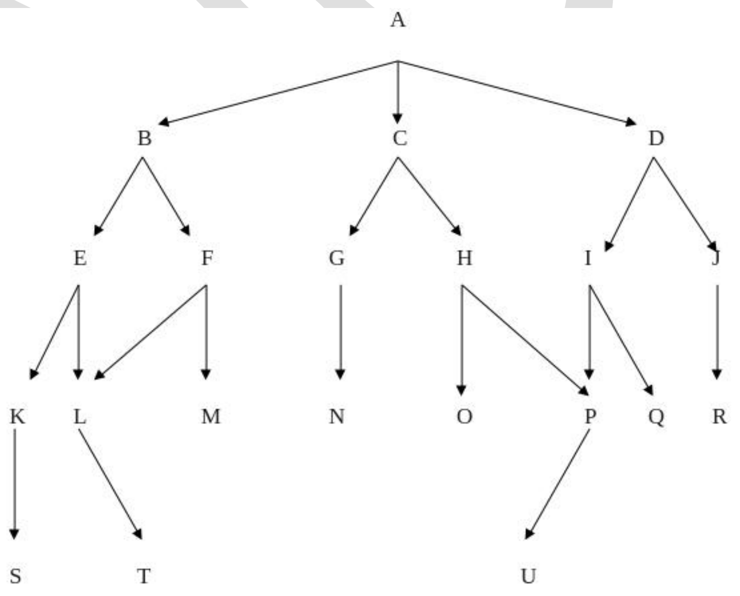
Open	Closed
A	empty
BCD	A
CDEF	BA
DEFGH	CBA
EFGHIJ	DCBA
FGHIJKL	EDCBA
GHIJKLM	FEDCBA
HIJKLMN	GFEDCBA
.	
.	
.	
.	
And so on until open = [ ].	

Since breadth first search considers every node at each level of the graph before going deeper in to the space, all states are first reached along the shortest path from the start state. Breadth first search is therefore guaranteed to find the shortest path from the start state to goal.

2. Depth first search

Dept first search goes deeper in to the search space whenever this is possible.

Consider the graph



Depth first search examines the states in the graph in the order

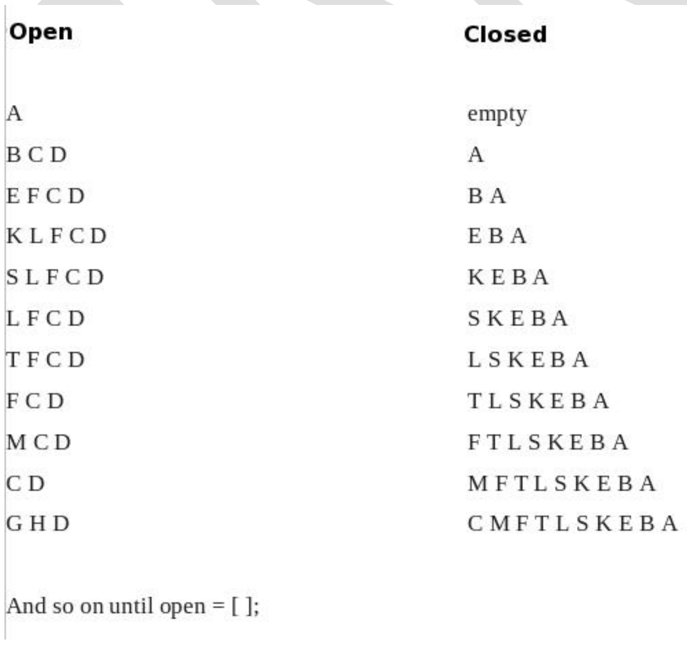
A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R.

In depth first search, the descendant states are added and removed from the left end of 'open'.

That is, 'open' is maintained as a stack (LIFO) data structure.

```
void depth_ first_search( )
{
   open = [ start ] ;
   closed = [ ] ;
   while ( open not empty )
   {
      Remove the leftmost state from open , call it X;
      if X is a goal ,
      then
         return SUCCESS;
      else
      {
         Generate children of X;
         Put X on closed ;
         Discard children of X, if already on open or closed ;
         Put remaining children on left end of open ;
      }
   }
   return FAIL ;
}
```

A trace of depth first search on the above graph is shown below.



Open	Closed
A	empty
B C D	A
E F C D	B A
K L F C D	E B A
S L F C D	K E B A
L F C D	S K E B A
T F C D	L S K E B A
F C D	T L S K E B A
M C D	F T L S K E B A
C D	M F T L S K E B A
G H D	C M F T L S K E B A

And so on until open = [ ];

'open' records all states that are discovered and 'closed' contains all states that are already considered.

DFS is not guaranteed to find the shortest path to a state the first time that state is encountered.

3. Depth limited search

To limit the depth of DFS, we can supply a predetermined depth limit, l to DFS. That is nodes at depth l are treated as if they have no successors. This approach is called depth limited search.

Depth first search can be viewed as a special case of depth limited search with I = 1.

Sometimes depth limit can be based on knowledge of the problem. For example, on the map of Romania, there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so I = 19 is a possible choice. But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the diameter of the state space, gives us a better depth limit , which leads to a more efficient depth limited search.

4. Uniform cost search

Breadth first search assumes that all path costs from a state to a successor is same. It expands the shallowest unexpanded node.

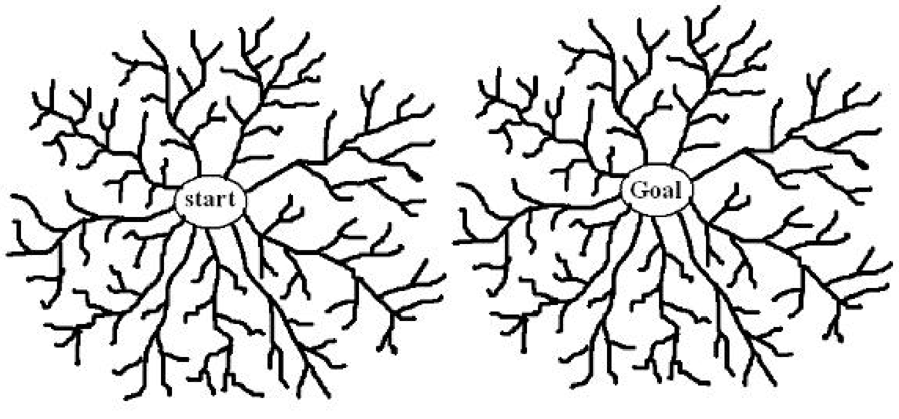
But uniform costs search expands the node n with the lowest path cost, instead of expanding the shallowest node.

Note that if all step costs are equal, this is identical to breadth first search.

Uniform cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if it ever expands a node that has a zero cost action leading back to the same state. We can guarantee completeness provided the cost of every step is greater than or equal to some small positive constant. The algorithm expands nodes in order of increasing path cost.

5. Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches- one forward from the initial state and other backward from the goal, stopping when the 2 searches meet in the middle.



SNGCE

A schematic view of a bidirectional search that is about to succeed , when a branch from the start node meets a branch from the goal node.

Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found. What do we mean by "the goal" in searching "backward from the goal".

For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search. If there are several explicitly listed goal states, then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states. The most difficult case for bidirectional search is when the goal test gives only an implicit description of some possibly large set of goal states. For example, all the states satisfying the "check mate" goal test in chess.

<u>Heuristic Search Technique</u>

Generate-and-Test Algorithm

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others it means generating a path from a start state.

2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states

3. If a solution has been found, quit. Otherwise return to step 1.

If the generation of possible solution is done systematically, then this procedure will find a solution eventually if one exists. If the problem space is very large eventually may be a very long time.

It is a depth first search procedure since complete solutions must be generated before they can be tested. It can also operate by generating solutions randomly but then there is no guarantee that a solution will ever be found. It is also known as **British Museum Algorithm,** a reference to a method for finding an object in the British museum by wandering randomly. Between these two extremes considered because they seem unlikely to leas to a solution.

The most straightforward way to implement systematic generate and test is a depth first search tree with backtracking. For simple problems exhaustive generate and test is often a reasonable technique.

For example, Consider the puzzle that consist of four six-sided cubes with each side of each cube painted one of the four colors. A solution to the puzzle consist of an arrangement of the cubes in a row such that on all four sides of the row one block face of each color is showing. This problem can be solved by a person in several minutes by systematically and exhaustively trying all possibilities. It can be solved even more quickly using a heuristic generate-and-test procedure.

A quick glance at the four blocks reveals that are more, say, red faces than there are of other colors. Thus when placing a block with several red faces, it would be a good idea to use as few of them as possible as outside faces. **Using this heuristics many configuration need never be explored and a solution can be found quite quickly.**

SNGCE

Unfortunetly for problems much harder than this even heuristic generate-and-test all by itself is not an a very effective technique. But when combined with other technique to restrict the space in which to search even further, the technique can be very effective.

Plan-Generate-Test

Consider an example AI program DENDRAL,

Dendral was a project in artificial intelligence (AI) of the 1960s, and the computer software expert system that it produced. Its primary aim was to infer the structure of organic components using mass spectogram and nuclear magnetic resonance data.

Dendral uses plan-generate test strategy in which a planing process that uses constraint-satisfaction techniques creates list of recommended and contraindicated substructures. The generate-and-test procedure then uses those lists so that it can explore only a fairly limited set of structures. Constrained in this way generate-and-test procedure has proved highly effective.

This combination of planning using one problem solving method with the use of plan by another problem solving method, generate-and-test is an excellent example of the way techniques can be combined to overcome the limitations that each processes individually.

A major weakness of planning is that it often produces somewhat inaccurate solutions since there is no feedback from the world. But by using it only produce pieces of solution that will then be exploited in the generate-and-test process, the lack of detailed accuracy become unimportant.

Best First Search

Best First Search is a way of combining the advantages of both depth first search and breadth first search method.

OR Graphs

Depth first search is good because it allows a solution to be found without all competing branches having to be expanded.

Breadth First Search is good because it does not get trapped in dead end paths. One way of combining the two is to follow a single path at a time, but to switch paths whenever some competing path looks more promising than the current one.

At each step of Best First Search we select the most promising of the nodes. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using rules to generate its successors. If one of them is a solution we can quit. If not all those new nodes are added to set of nodes generated so far. A bit of Depth First Searching occurs as the most promising branch is explored. If a solution is not found, that branch will start to be less promising than one of the top level branches that had been ignored. The old branch is not forgotten. Its last node remain in the set of generated but unexpected nodes.

SNGCE

Initially there is only one node, so it will be expanded. Doing so generate three new nodes. Heuristic function is estimate of the cost of getting to a solution from given node, is applied to each of these new nodes. Since D is most promising it is expanded next, producing successor nodes, E and F. But then the heuristic function is applied to them. Now another path going through node B looks more promising so it is pursued, generating nodes G and H. But again these nodes looks less promising than another path so attention is returned to path through D to E. E is expanded yielding nodes I and J. At the next step J will be expanded since it is most promising. This process continue till a solution is found.

Hill Climbing vs Best Fit

This procedure is very similar to steepest ascent hill climbing with two exceptions:

1. in hill climbing one move is selected all others are rejected never reconsidered

2. Best available state is selected in best fit search even if state has value lower than value of state just explored. Whereas hill climbing will stop if there is no successor states with better value than current state

Directed Graph Searching (OR Graph)

It is sometimes important to search a graph a graph instead of a tree so that duplicate paths will not be taken. An algorithm to do so will operate by searching a directed graph in which each node represent a point in problem space.

Each node will contains:

l addition to a description of problem state it represents,

l an indication of how promising it is,

l a parent link that points back to best node it came from, and

l a list of nodes that were generated from it

The parent link will make it possible to recover the path to goal once goal is found. The list of successors will make it possible if a better path is found to an already existing node, to propagate improvement down to its successors.

We call this an OR-Graph since each of this branch represents an alternate problem solving path.

To implement this we need two list of nodes

1. OPEN

Nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined(ie had their successors generated). OPEN is a priority queue in which the element with highest priority are those with most promising value of heuristic function.

2. CLOSED

SNGCE

Nodes that have already been examined. We need to keep these in memory if we want to search a graph rather than a tree, since whenever a new node is generated we need to check if it is generated before.

Heuristic Function

The heuristic function must be able to search more promising paths first. Call this function *f'* (approximation to a function).

We can call this function as sum of two components

l g

measure of cost of getting from initial state to current node.

It is not an estimation

It is exact sum of cost of applying each of the rules that were applied along the best path to the node

Must be non negative else graph will traverse in a cycles which will appear to get better but will be getting longer

l h'

It is an estimate of additional cost of getting from current node to goal state.

Good one get low value, bad one get high value

l f' = g + h'

F' represents an estimate of cost of getting from initial state to goal state along the path generated from current node

If more than one path generated the node, the best path is record

Actual Operation

l Proceed in steps, expanding each node at each step, until it generate a goal state node

l At each step pick the most promising node that have so far been generated but not expanded.

n It generate successor of the chosen node

n Apply heuristic function to them

n Add them to list of open nodes, after checking to see if any of them have been generated before. By doing above check we guarantee that each node appear exactly once in the graph, although many nodes may point to its successors

<u>Algorithm Best First Search</u>

SNGCE

1. Start OPEN containing initial state

2. Until goal is found or there are no nodes present in OPEN do:

    a) Pick up the best node on OPEN

    b) Generate its successors

    c) For each successor do

        i. If it has not been generated before, evaluate it add it to open, record its parent

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

The A* Algorithm

The best first search algorithm is a simplification of A* algorithm. This algorithm uses the same heuristic functions f', g and h' and also the list OPEN and CLOSED.

Algorithm A*

1. Start with OPEN containing only the initial node. Set that nodes g value to 0, and its h' value to whatever it is. And its f' value to h'+0 ie h'. Set CLOSED to an empty list

2. Until a goal node is found, repeat the following procedure:

If there is no node on OPEN, report failure

Otherwise pick a node on OPEN with lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED.

See if BESTNODE is goal node. If so exit and report solution.

Otherwise generate successor of BESTNODE but do not set BESTNODE to point to them yet.

For each such SUCCESOR do the following:

a) Set SUCCESOR to point back to BESTNODE. These backward links will make it possible to recover the path once a solution is found.

b) Compute g(SUCCESOR) = g(BESTNODE) + cost of getting from BESTNODE to SUCCESOR

c) See if successor is same as any node in OPEN

If so call that node OLD throw SUCCESOR away and add OLD to list of BESTNODE's successors. Now we must decide whether OLDS's parent link should be reset to point to BESTNODE if path we have just found is cheaper than current best path to OLD (since SUCCESOR and OLD are same nodes)

SNGCE

So whether it is cheaper to get to OLD via its current parent or to SUCCESOR via BESTNODE comparing their g values. If OLD is cheaper then do nothing. If SUCCESOR then reset OLD's parent link to point to BESTNODE, record new cheaper path in g(OLD), and update f'(OLD).

d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so so call the node on CLOSED OLD node and add it to list of BESTNODES successors.

Check to see if new path or old path is better and set parent link and g and f' function accordingly.

If we have found a better path to CLOSED OLD we must propagate improvements to OLD's Successor. This is a tricky process.

OLD points to its successors. Each successor in turn points to its Successors and so forth until each branch terminates with a node that is either is still on OPEN or has no successors.

while propagation check if path through current parent is still better than new path then stop propagation

e) if SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN add its to list of BESTNODES's successor.

Compute f'(SUCCESSOR)=g(SUCCESSOR)+h'(SUCCESSOR)

Problem Reduction

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

AND-OR Graphs

AND-OR graph is useful for representing the solution of problem that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution. Just as in an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the problem might be solved. This structure is an AND-OR Graph.

AND arcs are indicated with a line connecting all the components.

In order to find solutions in an AND-OR graph we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately. This algorithm should find a path from starting node of the graph to a set of nodes

SNGCE

representing solution state. It may be necessary to get more than one solution state since each arm of an AND arc must lead to its own solution node.

The top node A has been expanded, producing two arcs one leading to B and one leading to C and D. the number at each node

AO* Algorithm

The AO* algorithm uses a single structure GRAPH, representing the part of the search graph that has been explicitly generated so far. Each node in the graph will point both down to its immediate successors and up to its immediate predecessors.

Each node in the graph will also have associated with it an h' value, an estimate of cost of a path from itself to a set of solution nodes. We will not store g as we did in A* algorithm. It is not possible to compute a single such value since there may be many path to the same state. And such value is not necessary because of the top down traversing of the best known path.

Algorithm AO*

1. Let GRAPH consist only of the node representing the initial state. Call this node INIT. Compute h'(INIT).
2. Until INIT is labelled SOLVED or until INIT's h' value become greater than FUTILITY, repeat the following procedure:
    a. trace the labeled arcs from INIT and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call this the selected node NODE
    b. Generate the successor of NODE. If there are none, then assign FUTILITY as the h' value of NODE. **This is equivalent to saying that NODE is not solvable**. If there are successors, then for each one that is not also an ancestor of NODE do the following:
        i. Add SUCCESSOR to the GRAPH
        ii. If the SUCCESSOR is a terminal node, label it SOLVED and assign it an h' value of 0
        iii. If SUCCESSOR is not a terminal node, compute its h' value
    c. Propagate the newly discovered information up the graph by doing the following. Let **S be a set of nodes that have been labelled SOLVED** or whose h' value have been changed and so need to have values propagated back to their parents. Initialize S to NODE. Until S is empty, repeat the following procedures
        i. If possible, select from S a node, none of whose descendants in the GRAPH occurs in S. If there is no such node, select any node from S. Call this node CURRENT and remove it from S
        ii. Compute the cost of each of the arcs emerging from CURRENT. The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the

arc itself is. Assign as CURRENT's new h' value the minimum of costs just computed for the arcs emerging from it.

iii. Mark the best path out of current by marking the arc that had the minimum cost as computed in previous step.

iv. Mark CURRENT as SOLVED if all of the nodes connected to it through the new labeled arc have been labelled SOLVED.

v. If CURRENT has been labelled SOLVED or if the cost of CURRENT was changed then its new status must be propagated back up the GRAPH. So add all of the ancestors of CURRENT to S.

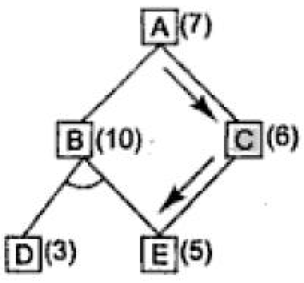


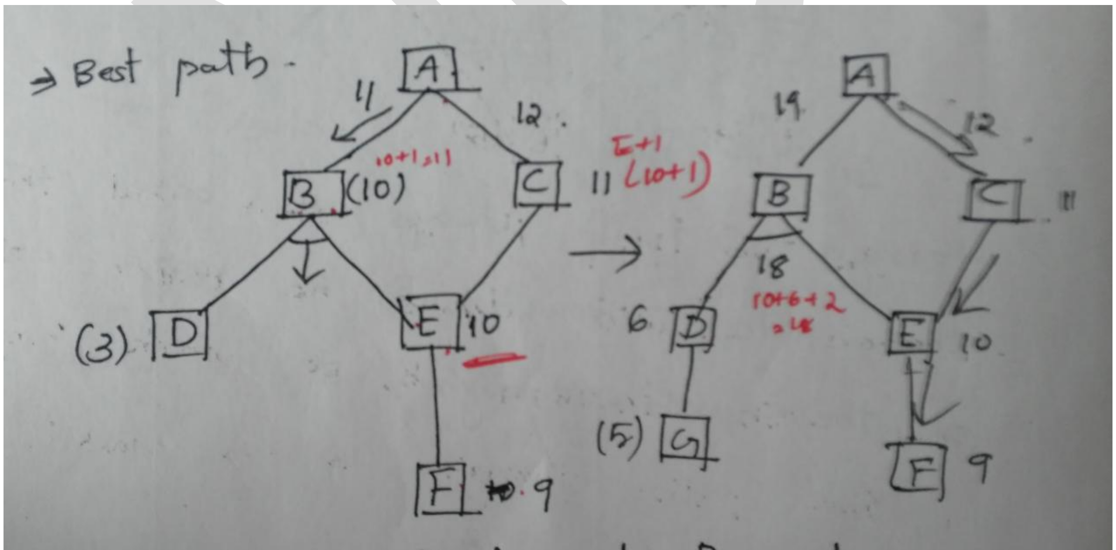
Fig. 3.11 *An Unnecessary Backward Propagation*

In this figure it is clear that the path through C will always be better than path through B, so work expanded on the path through B is wasted.

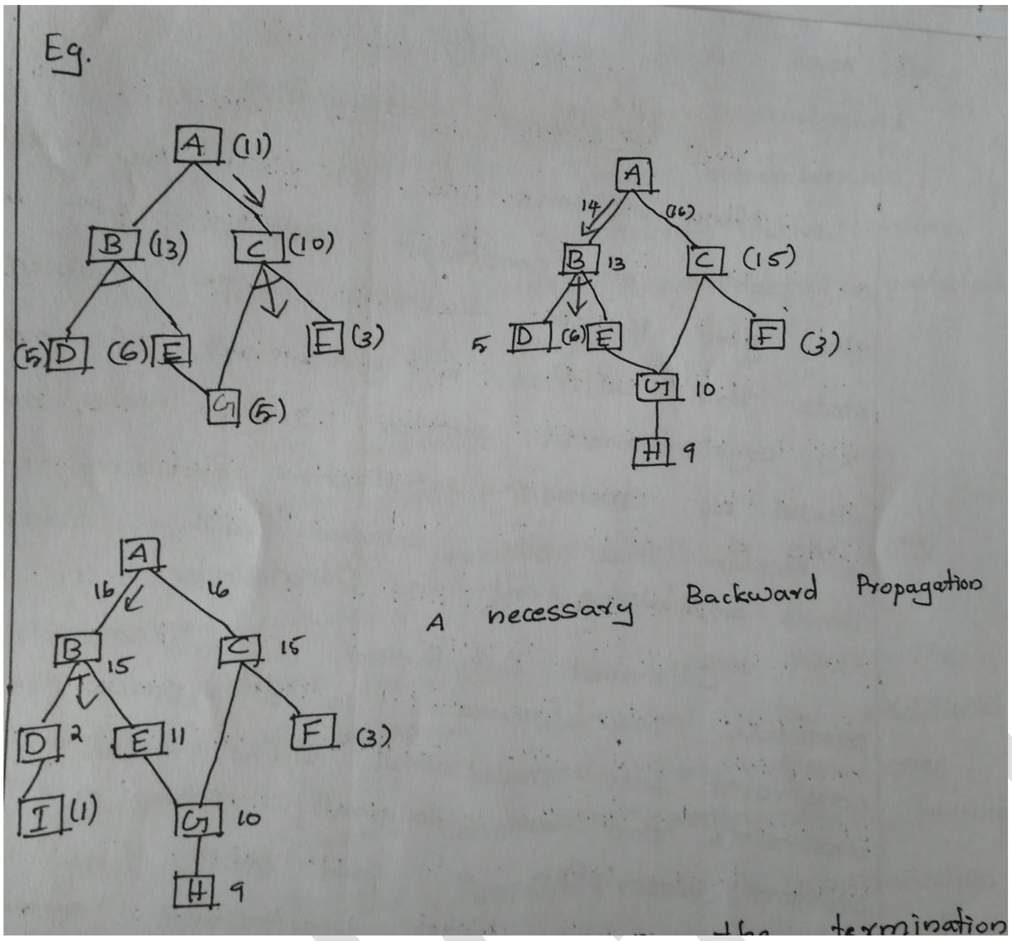
But if the cost of E is revised and that change is not propagated up through C, B may appear to be better.

For example if as a result of expanding node E, we update its cost to 10, Cost of C will be updated to 11.

if this is all that is done then when A is examined the path through B will have a cost of only 11 compared to 11 for path through C and it will be labelled erroneously as the most promising path. The mistake might be detected at the next step, during which D will be expanded. If its cost changes and is propagated back to B then B's cost will be recomputed and the new cost of E will be used. Then the new cost of B will be propagated back to A. At that point through C will again be better.



An unnecessary Backward Propagation.



A second point concern the termination of backward cost propagation. Because GRAPH may contain cycles there is no guarantee that this process will terminate simply because it reaches the top of the graph

Constraint Satisfaction

Many problems in AI can be viewed as problems of constraint satisfaction in which the goal is to **discover some problem state that satisfies a given set of constraints**.

eg, crypt arithmetic puzzles. Design tasks can be viewed as constraint satisfaction problems in which a design must be created within fixed limit on cost, time and materials.

Constraint satisfaction is a search procedure that operates in a space of constraint sets.

Initial state: contains the constraint that are originally given in the problem description.

Goal state: is any state that has been constrained "enough" where "enough" must be defined for each problem. For example for arithmetic enough means that each letter has been assigned to a unique numeric value.

Constraint satisfaction is a two step process.

SNGCE

1. First, constraints are discovered or propagated as far as possible throughout the system. Then if there is still not a solution search begins.
2. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint.

The first step, propagation arises from the fact that there are usually dependencies among the constraints. These dependencies occurs because many constraints involves more than one object and many object participate in more than one constraint.

eg, one constraint , N=E+1

added constraint N=3

We could propagate that to get a stronger constraint on E, E=2

Constraint propagation also arise from the presence of inference rules that allow additional constraint to be inferred from the ones that are given. Constraint propagation terminates for one of the two reasons:

1. A contradiction may be detected. If the contradiction involves only those constraints that were given as part of problem specification then no solution exists.
2. A propagation has run out of stream and there are no further changes that can be made on the basis of current knowledge.

At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made. In the case of arithmetic problem, this actually means guessing a particular value for some letter.

If a contradiction is detected then backtracking can be used to try a different guess and proceed with it.

Algorithm: Constraint Satisfaction

1. Propagate available constraints. to do this first set OPEN to set of all objects that must have values assigned to them n a complete solution. Then do until an inconsistency is detected or until OPEN is empty.
    a. Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.
    b. If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraint with OB
    c. Remove OB from OPEN
2. If the union of constraint discovered above defines a solution, then quit and report the solution.
3. if the union of the constraints discovered above define a contradiction, then return failure .
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this loop until a solution is found or all possible solution have been eliminated.
    a. Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.

SNGCE

   b. recursively invoke constraint satisfaction with the current set of constraints augmented by strengthening constraint just selected

For eg, Consider the arithmetic problem. The goal state is a problem state in which all letters have been assigned a digit in such a way that all initial constraints are satisfied.

Problem

```
                S    E    N    D

                                          +M    O    R    E

                                          -----------------------

                                         M    O    N    E    Y
```

Initial State: No two letter have same value. the sum of digits must be as shown in problem.

The solution process proceeds in cycles. At each cycle, two significant things are done.

1. Constraints are propagated by using rules that corresponds to properties of arithmetic
2. A value is guessed for some letter whose value is not yet determined. A few useful heuristic can help to select the best guess to try first. For eg, if there is a letter that has only two possible values and other with six possible values there is a better chance of guessing right on first than second.
   a. if there is a letter that participate in many constraints then it is good idea to prefer it to a letter that participate in a few

Let C1, C2, C3 and C4 indicate the carry bits. Initially rules for propagating constraints generate the following additional constraints:

- M=1, Since two single digit numbers plus a carry cannot total more than 19 (9+9+1)
- S=8,9. Since S+M+C3 > 9 (to generate carry)
  - M=1
  - S+1+C3>9
  - therefore S+C3>8
  - C3 can be at most be 1
- O =0
  - S+M+C3 must be at least 10 to generate a carry and it can be at most 11
  - S + M + C3 -> MO
  - => S+1+C3
  - =>(8/9) +1 + (1/0)
  - Ie either 8+1+1=10
  - or 8+1+0=9   this is not possible should be 2 digit to have carry
  - or 9+1+1=11
  - or 9+1+0 =10
- But M is already 1 so O must be 0.
- N=E or E+1 depending value of C2
  - C2+E+O -> C3N
  - C2 + E ->C3N
  - Eg, E=9   C2=1 or 0

SNGCE

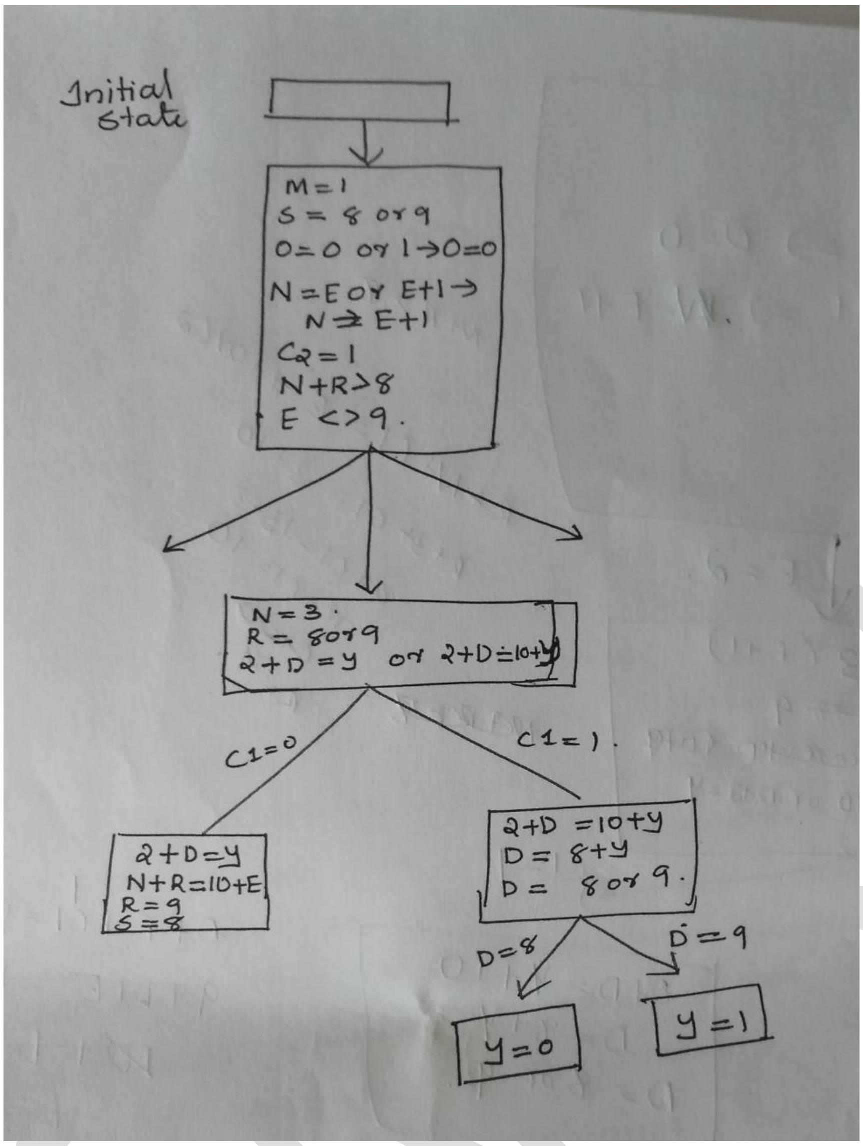
  - ○ E be any value [0-9]
  - ○ We consider 9 as E
  - ○ C2+9 -> C3N
  - ○ (0/1) +9 -> C3N
  - ○ If C2is 1 and E is 9 then the value of N is 0. ie not possible. Already O=0.
  - ○ Therefore C3=0 and N=E or E+1
  - ○ But N=E is not possible
  - ○ so N=E+1 and C2=1
- In order for C2 to be 1, the sum of N+R+C1 must be greater than 9, so N+R must be greater than 8
- N+R cannot be greater than 18, even with a carry in so E cannot be 9

At this point let us assume that no more constraints can be generated. Then to make progress from here , we must guess. Suppose E is assigned the value 2.

  - N=3 since N=E+1
  - R=8 or 9, Since R+N(3)+C1(0/1)=2 or 12
  - But since N is already 3, the sum of these non negative numbers cannot be less than 3
    - ○ R+3+(0/1)=12
    - ○ R=8 or 9
  - 2+D=Y+10

Suppose C1 is chosen to guess a value. If we try 1 then we eventually reach dead end. When this happens the process will backtrack and try C1=0.

SNGCE



Crypt Arithmetic Problems

Crypt Arithmetic or verbal arithmetic is a class of mathematical puzzles in which the digits are replaced by letters of the alphabet or other symbols. Usually it is required that each letter would be replaced by a unique digit. Each letter having different value from the other letters. Alphametic problem is a subset of CryptArithemetic where the arithmetic operation is summation. This problem is under category of Constraint Satisfaction Problem.

It is an arithmetic problem which is represented in letters. It involves the decoding of digit represented by a character. It is in the form of some arithmetic equation where digits are distinctly represented by some characters. The problem requires finding of the digit represented by each character. Assign a decimal digit to each of the letters in such a way that the answer to the problem is correct. If the same letter occurs more than once, it must be assigned the same digit each time. No two different letters may be assigned the same digit

Procedure

- Cryptarithmatic problem is an interesting constraint satisfaction problem for which different algorithms have been developed. Cryptarithm is a mathematical puzzle in which digits are replaced by letters of the alphabet or other symbols. Cryptarithmatic is the science and art of creating and solving cryptarithms.
- The different constraints of defining a cryptarithmatic problem are as follows.
    1. Each letter or symbol represented only one and a unique digit throughout the problem.
    2. When the digits replace letters or symbols, the resultant arithmetical operation must be correct.
- The above two constraints lead to some other restrictions in the problem.

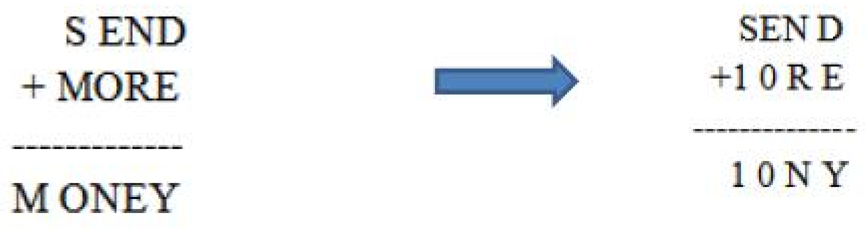
Example

Consider that, the base of the number is 10. Then there must be at most 10 unique symbols or letters in the problem. Otherwise, it would not possible to assign a unique digit to unique letter or symbol in the problem. To be semantically meaningful, a number must not begin with a 0. So, the letters at the beginning of each number should not correspond to 0. Also one can solve the problem by a simple blind search. But a rule based searching technique can provide the solution in minimum time.

```
 .EAT
+THAT
------
APPLE
```

1. Step 1:

   In the above problem, M must be 1. You can visualize that, this is an addition problem. The sum of two four digit numbers cannot be more than 10,000. Also M cannot be zero according to the rules, since it is the first letter.



2. Step 2

   Now in the column s10, s+1 ≥ 10. S must be 8 because there is a 1 carried over from the column EON or 9. O must be 0 (if s=8 and there is a 1 carried or s = 9 and there is no 1 carried) or 1 (if s=9 and there is a 1 carried). But 1 is already taken, so O must be 0.

```
  S END
+ 1 0 RE
-----------
 1 0 NEY
```

3. Step 3

   There cannot be carry from column EON because any digit +0 < 10, unless there is a carry from the column NRE, and E=9; But this cannot be the case because then N would be 0 and 0 is already taken. So E < 9 and there is no carry from this column. Therefore S=9 because 9+1=10.

4. Step 4

   In the column EON, E cannot be equal to N. So there must be carry from the column NRE; E+1=N. We now look at the column NRE, we know that E+1=N. Since we know that carry from this column, N+R=1E (if there is no carry from the column DEY) or N+R+1=1E (if there is a carry from the column DEY).

```
   9 E N D
 + 1 O R E
------------------
 1 O N E Y
```

   Let us see both the cases:

   - ➢ No carry: N +R =10 +(N- 1)= N + 9
   - ➢ R = 9

   - ● But 9 is already taken, so this will not work – Carry: N +R + 1= 9 – R = 9 – 1= 8 This must be solution for R.

```
   9 E N D
 + 1 O R E
------------------
 1 O N E Y
```

5. Step 5

   Now just think what are the digits we have left? They are 7, 6, 5, 4, 3 and 2. We know there must be a carry from the column DEY. So D+E>10.N = E+1, So E cannot be 7 because then N would be 8 which is already taken. D is almost 7, so E cannot be 2 because then D +E <10 and E cannot be 3 because then D+E=10 and Y=0, but 0 is already taken. Also E cannot be 4 because if D >6,D+E<10 and if D=6 or D=7 then Y=0 or Y=1, which are both taken. So E is 5 or 6.

SNGCE

If E=6, then D=7 and Y=3. So this part will work but look the column N8E. Point that there is a carry from the column D5Y.N+8+1=16(As there is a carry from this column). But then N=7 and 7 is taken by D therefore E=5.

```
   9 5 N D
 + 1 0 8 5
-------------------
 1 0 N 5 Y
```

6. Step 6

   Now we have gotten this important digit, it gets much simpler from here N+8+1=15, N=6

```
   9 5 6 D
 + 1 0 8 5
-------------------
 1 0 6 5 Y
```

7. Step 7

   The digits left are 7, 4, 3 and 2. We know there is carry from the column D5Y, so the only pair that works is D=7 and Y= 2. Which is final solution of this problem.

```
   9567
 + 1085
-------------------
 1 0652
```

SNGCE