Heuristic Search

Unit 2

Brandon Syiem

Heuristic Search Methods

Sometimes called *weak methods* - as they are unable to solve problems without exploiting knowledge.

- Generate-and-Test
- Hill Climbing (and its variants)
- Simulated annealing
- Best-First Search
- OR-Graphs
- A* algorithm
- Problem Reduction
- AND-OR Graphs
- AO* algorithm
- Constraint Satisfaction
- Means End Analysis

Generate-and-Test Technique

1. Set current state as initial state
2. Generate any *possible solution* (may be path to a goal state or a goal state)
3. Check to see if the solution state matches any of the goal states
4. If the solution state is a goal state
    a. Quit
5. Else
    a. Go to step 2

Possible solution - may check every node, if it is a goal state (water jug problem)
- OR, may require certain conditions to be satisfied (complete solutions)
(travelling salesman problem using array to represent path -
Condition may be that array is full, i.e, all cities have been visited)

Cont ..

- Generate-and-Test is a depth-first search procedure
- If done systematically, will eventually find a solution but can be inefficient
- Can be done randomly, but no guarantee of finding a solution
  - Called **British Museum algorithm**
- Can utilize a heuristic to disregard unlikely paths while operating systematically.
- Can be done for search trees or search graphs

Read ***Plan-Generate-and-Test*** and an example of generate-and-test from *Rich & Knight* pg. 51

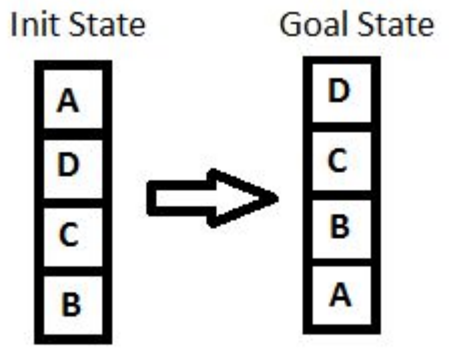
Hill Climbing

- Generate-and-test + heuristic

Simple Hill Climbing Algorithm:

- Set current state(***cs***) as initial state
- Check to see if current state is goal state, if yes, quit.
- Else, Loop
    - Apply a legal operation (that has not been previously applied) to the ***cs*** to create a new state(***ns***).
    - Check to see if ***ns*** $\in$ goal states
        - If yes, quit
    - Else, check if ***ns*** is "better" than the current state using a heuristic function
    - If ***ns*** is better than ***cs***,
        - ***cs = ns***

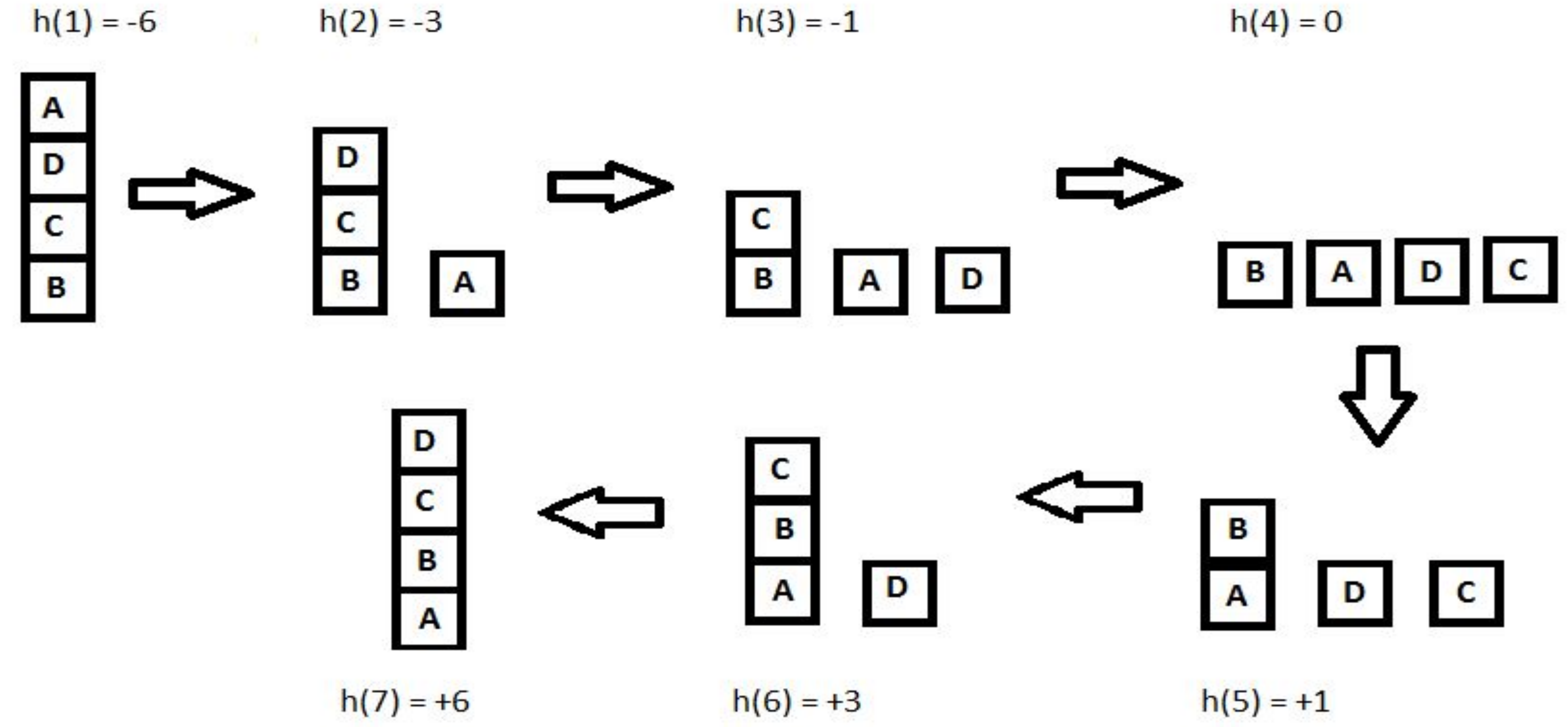
Hill Climbing Example



Given a set of blocks, we wish to stack them according to the picture.

Lets employ a simple heuristic:

h(x) = +1 for all blocks in the support structure, if a block (***b***) is correctly positioned
(i.e, all blocks stacked below a ***b*** should be correct).
-1 for all blocks in the support structure, if a block (***b***) is incorrectly positioned.
(i.e, if any block stacked below a ***b*** is incorrect).



http://www.baeldung.com/java-hill-climbing-algorithm

What if?

We employ a heuristic function such that

h(x) = +1 for every block that is resting on the thing it is supposed to be resting on
-1 for every block that is resting on something it is **not** supposed to be resting on

Steepest-Ascent Hill Climbing

- Similar to Simple Hill Climbing but differs in the way the next node is chosen
- Simple Hill Climbing updates current state to the first neighbouring state it finds that is better than the current state, Steepest-Ascent Hill Climbing considers all neighbouring states and sets the current state to the best neighbouring state.

Cont ..

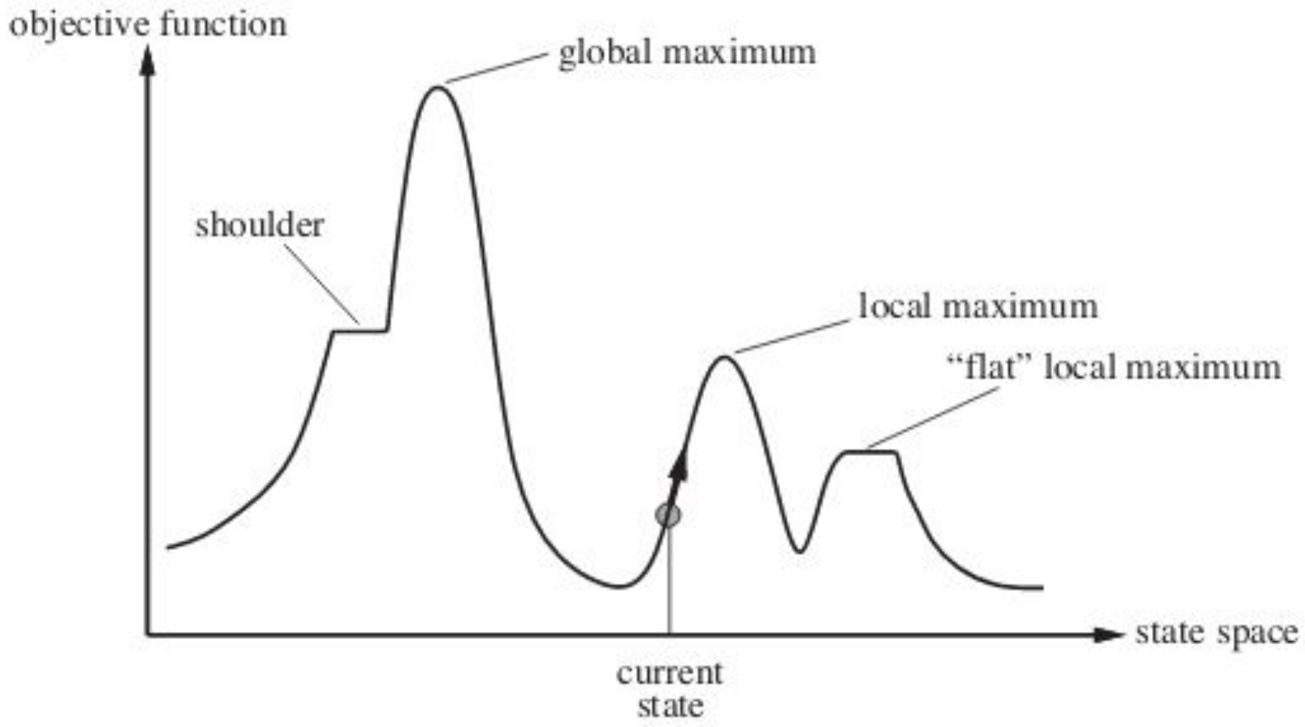
Algorithm:

- Set current state (***cs***) as initial state
- If ***cs*** is goal state, quit
- Else, loop
    - Set ***TARGET*** to a state such that any neighbouring state(successor) will be better than target
        - Evaluation of a state to be “better” is done by a heuristic function (***h()***)
    - For each action applicable to current state
        - Apply the action to create a new state (***ns***)
        - If ***ns*** is a goal state, quit
        - Else, Evaluate ***h(ns)***,
        - If ***h(ns)*** is better than ***h(TARGET)***
            - ***TARGET = ns***
    - If ***h(TARGET)*** is better than ***h(cs)***
        - ***cs = TARGET***

Disadvantages

Simple hill climbing and Steepest-Ascent hill climbing:

- Both variations may fail to arrive at a goal state by getting stuck at a point where no better states are reachable


objective function
global maximum
shoulder
local maximum
"flat" local maximum
state space
current state

Cont ..

- **Local Maximum**: a state that is better than all its neighbours but may not be better than some states further away.
    - Utilize backtracking to overcome this issue
- **Plateau:** a flat region, all neighbouring states have the same values to the heuristic function.
    - Not sure which direction to move, Randomly jump to a state far away from the current state
- **Ridge:** It is region which is higher than its neighbours but itself has a slope (uphill). It is a special kind of local maximum.
    - To overcome, use two or more moves (actions) and then test to see if we are moving in the desired direction

Simulated annealing

- Named after the annealing process (metallurgy)
- Used to handle plateau, ridge and local maxima problems faced by simple and steepest-ascent hill climbing.
- Basic Idea
  - When a material is heated (to make it easier to work with) and then cooled to reach a stable state
  - Naturally goes from high energy to low energy state but there is a chance of going from low energy to high energy
  - Easier to go to higher energy state when temperature is higher.

$$p = e^{-\Delta E/kT}$$

Cont..

- p = probability of going to higher energy state
- ΔE = positive change in energy
- k = boltzman's constant
- T = Temperature

The rate at which the material is cooled has an adverse effect on the material

- Cooling too fast increases chances of stable high energy zones.
- Cooling too slow wastes time.
- The rate at which the Temperature is reduced is called the ***annealing schedule***

Cont ..

- Heuristic function is called the objective function
- Objective function aims to minimize rather than maximize (returns energy value)
- ΔE = (current state energy value) - (new state energy value)
- Keep track of Best-so-far solution incase we end up at a worse state
- Incorporate boltzman's constant (k) into Temperature (T) in the simulated annealing algorithm,
    - $kT = T$

$$p' = e^{-\Delta E/T}$$

Simulated Annealing Algorithm

- Check if initial state is goal, if yes, quit. Else set current state (***cs***) to initial state
- BEST-SO-FAR (BSF) = **cs**
- Initialize T according to a annealing schedule
- Loop until goal or no solution is found
    - Apply an action that has not been applied yet to **cs** to get a new state **ns**
    - Compute ΔE = (**value of cs) - (value of ns)**
    - If **ns** is goal state then quit
    - If **(value of ns)** < **(value of cs),**
        - **cs** = **ns**
        - **BSF** = **ns**
    - If **(value of ns)** > **(value of cs),**
        - Calculate ***p’***, Generate a random number (**n**) in the range [0,1]
        - If **p’** > **n**
            - Then **cs** = **ns**

Cont ..

    - Revise T according to schedule
- Return **BSF**

Example: http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6

Annealing Schedule

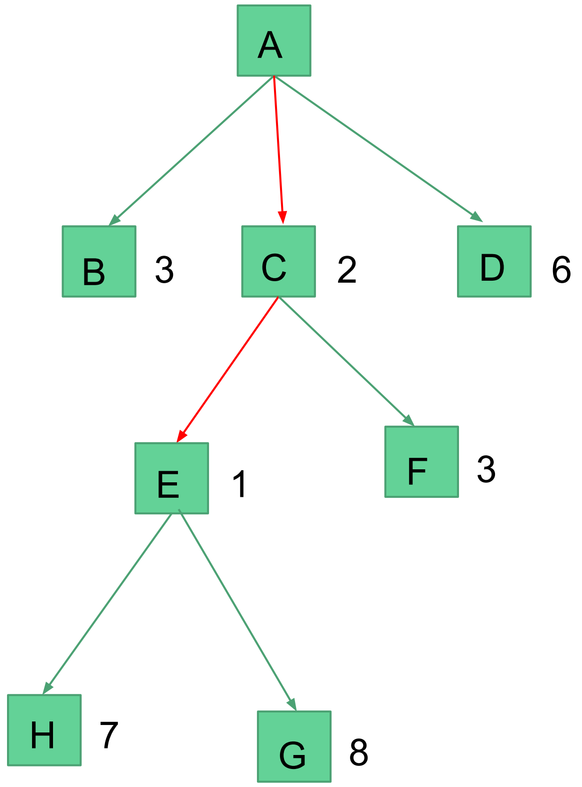
Has three components

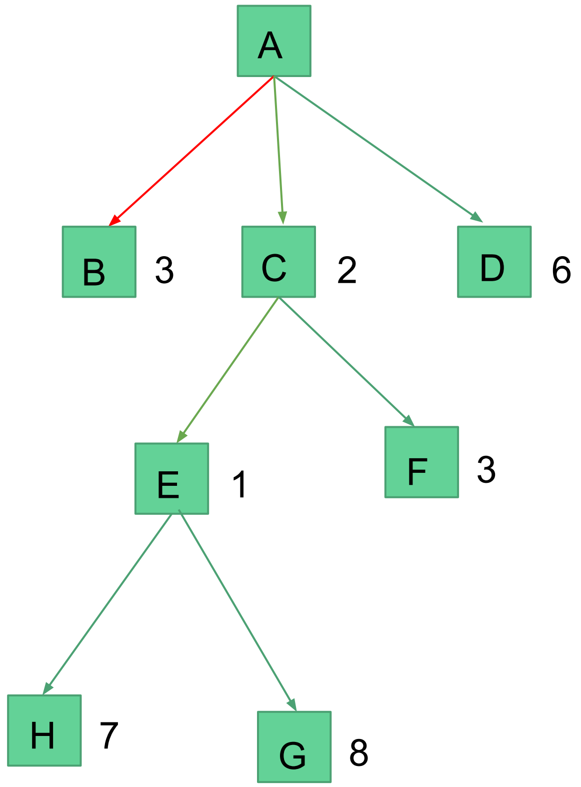
- The initial value of T
- When to change T
- By how much do we reduce T

**Usually the annealing schedule is determined empirically

Best-First Search (BestFS)

- Combines the advantage of DFS and BFS
- Uses priority queues
- Selects most promising node from queue (using some heuristic function)
- Expands the most promising node and applies heuristic function to children
- Inserts children to priority queue (priority depending on heuristic function value)
- **Important:** Always selects most promising node/state even if that node has “worse” h() value than the current state


A
B
3
C
2
D
6
E
1
F
3
H
7
G
8


A
B
3
C
2
D
6
E
1
F
3
H
7
G
8

OR-Graphs

- BestFS for directed graph traversal
- Makes use of two node_lists
    - OPEN list which is the priority queue
    - CLOSED list or the list of visited nodes
- Each node stores,
    - State information
    - Heuristic function value
    - Parent link - to determine best path to the current node
    - Successor node link - stores generated successor/ children nodes (to propagate changes in paths to current node to its generated successors).

Cont ..

- Uses a function ***f'***
- ***f' = g + h'***
    - ***f' =*** *is our heuristic function (estimation of cost from initial to goal state).*
    - ***g*** *= is the cost from initial state to current state.*
    - ***h'*** *= is the estimated cost from current node to the goal state.*
- **g** cannot be negative

OR-Graph algorithm (Best First Search - graphs)

- Put initial state into OPEN
- Loop until goal or no more nodes to expand to
    - **cs** = Dequeue OPEN
    - Generate children of **cs**
    - For each child of cs
        - IF child is not in CLOSED/OPEN,
            - Evaluate child using **f’** and record its parent
            - Enqueue child (note: since OPEN is a priority queue, they will be ordered depending on the value of **f’**)
        - IF child is in CLOSED/OPEN
            - IF the new path we just calculated is better than the previous path of the child
                - Update the parent, cost to get to the node and propagate changes to child’s children if any.

A* Algorithm

- The BestFS is a simplification of the A* algorithm
- Uses **f’, g** and **h’**
- Uses OPEN and CLOSED lists
- First presented in *Hart et al [1968; 1972]*

A* Algorithm

- OPEN.enqueue(initial node) -> g = 0 and h’ is calculated
- CLOSED = new Queue()
- Loop
    - If (OPEN.isEmpty())
        - Return fail
    - BESTNODE = OPEN.dequeue()
    - CLOSED.add(BESTNODE)
    - If BESTNODE == goal
        - Return BESTNODE
    - Generate Children/Successors of BESTNODE

Cont ..

- For each Child C of BESTNODE
  - Set parent link to BESTNODE
  - g(C) = g(BESTNODE) + cost to get from BESTNODE to C
  - If C is in OPEN
    - Call the node in OPEN as OLD
    - Insert OLD to successor list of BESTNODE and delete C
    - Check to see if the **g** currently stored in OLD is better than the **g** we just calculated (via parent BESTNODE)
      - If **NO**,
        - Change the parent link of OLD to BESTNODE and update OLD’s **g** and **f’**

Cont ..

- If C is in CLOSED
  - Call node in CLOSED as OLD
  - Insert OLD to successor list of BESTNODE and delete C
  - Check to see if the **g** currently stored in OLD is better than the **g** we just calculated (via parent BESTNODE)
    - If **NO**,
      - Change the parent link of OLD to BESTNODE and update OLD’s **g** and **f’**
      - **Propagate Changes to OLD’s Children/Successor**

Propagation of changes

- Do a Depth-First Traversal with root = OLD
- Check if parent link at each node **N** of the traversal is the same as the node we just came from (Or whether the parent link can be traced back to OLD)
- If Yes,
    - Change **f’** and **g** appropriately
- Else
    - Check to see if the **g** currently stored in **N** is better than the **g** calculated from the path from **OLD.**
        - If NO,
            - Change **f’, g** appropriately and update **parent link** to point to the node we just came from (descendant of OLD)

Cont ..

- If C is not in CLOSED or OPEN
  - Compute f’(C) = g(C) + h’(C)
  - Put C in OPEN (based on **f’**)
  - Insert C in BESTNODES children/successor list.

Notes on A*

- If we only care about the final state
  - Set g = 0 for all actions
- If we want the fewest steps to a solution state
  - Set g to a constant (usually 1)
- If **h’** is a perfect function or **h’ = h**
  - A* returns the optimal path

Cont ..

- If **h’ = 0**
  - **g** controls the search
  - If **g** = 0
    - Search is **random**
  - If **g** = 1
    - Search is **Breadth-first traversal**

Cont ..

- If **h’** underestimates **h** (i.e., value of **h’** estimates a smaller value than the true function **h**)
  - We may waste effort looking at unfavourable paths.
- If **h’** overestimates **h** (i.e., value of **h’** estimates a larger value than the true function **h**)
  - We may find a non-optimal solution.
- **Graceful Decay of Admissibility**
  - Pg. 60 - 62 book

Problem Reduction

- OR graphs/Trees find a single solution to the goal. Single branch from initial node to goal node.
- Reduce or decompose a problem into smaller problems and then solve each of them to arrive at a goal (AND)
    - Represented by arcs called AND arcs
- Many Arcs may be present and this may lead to different ways of solving the problem (OR)

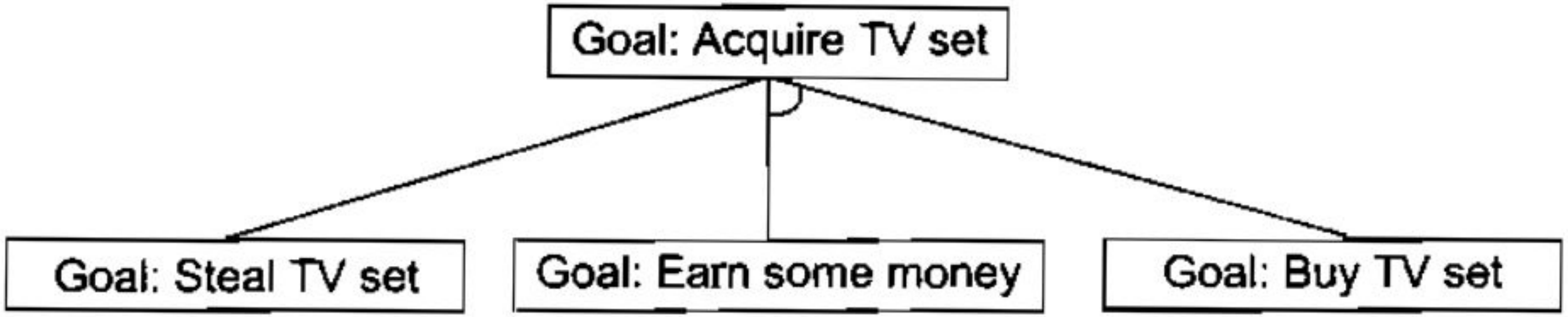


Fig. 3.6 *A Simple AND-OR Graph*

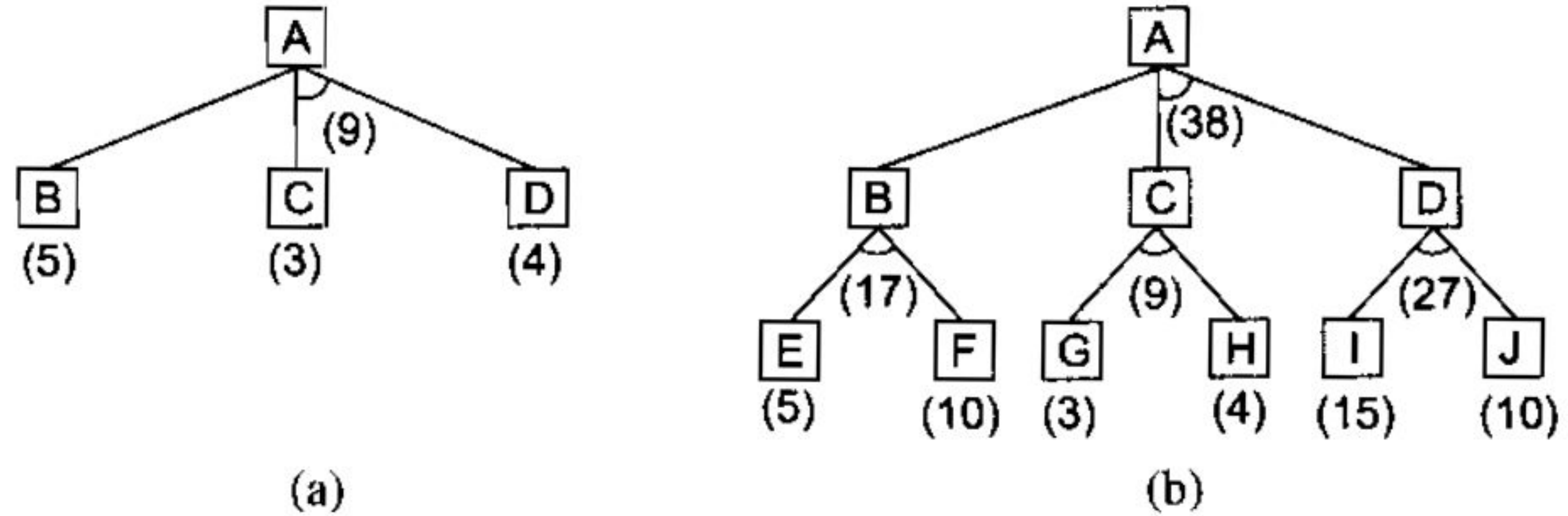


Fig. 3.7 *AND-OR Graphs*

AND-OR algorithm (Problem Reduction)

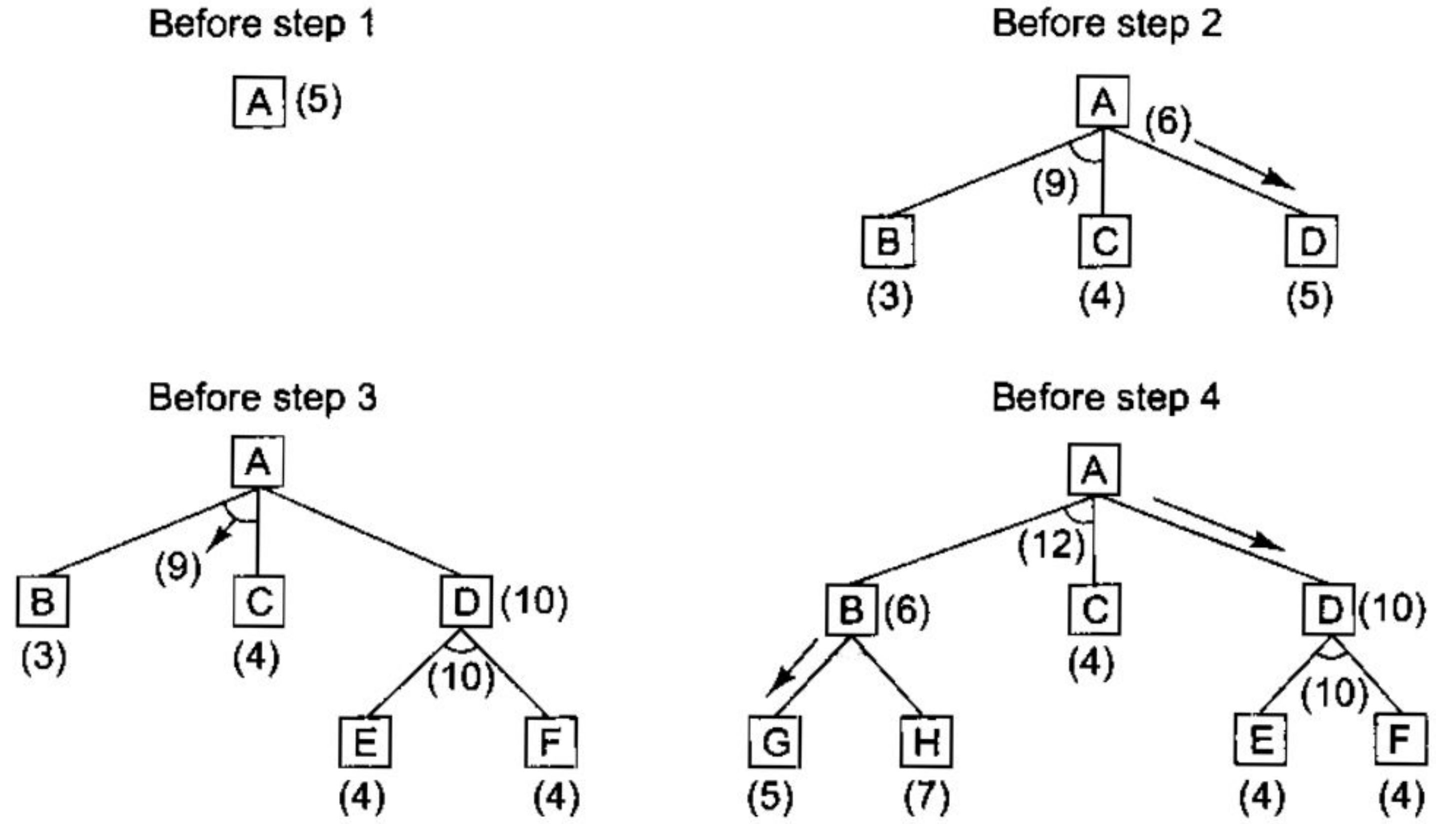
- Similar to best first search but differs in the sense that best path may contain AND arcs (where all the branches pointed to by the AND arc must be solved)
- The solution may be any number of states or path to those state.
- Uses a threshold, called the FUTILITY threshold.
    - If estimated cost of a path to a solution is more than FUTILITY, then we abandon the search.

Problem Reduction (AND-OR graph) Algorithm

- Initialize the graph to the starting node
- Loop until (starting node is SOLVED or its cost exceeds FUTILITY)
    - Traverse the graph starting from the start node and following the ***current best path*** and store the nodes that have not been expanded or solved.
    - Select a node (**N**) and expand it (Children C[ ])
        - If **N** has no children
            - Assign its value as FUTILITY.
        - If **N** has successors
            - Insert them into the graph and calculate their **h’**
            - If **h’** = 0 for any **C**
                - Label **C** as SOLVED.

Cont ..

- Change the **h’** of **N** to reflect the **h’** values of its children **C[ ]**
  - If any arc from N have all their nodes as SOLVED, then **N** is also SOLVED
  - else , set **h’** of **N** to the minimum **h’** of its arcs + cost of the arc.
  - Set **current best path** to arc that has minimum value (or SOLVED).
- Propagate these changes back in the graph
- When backtracking,
  - if any node has **all** the children of **any** of its arcs SOLVED
    - Then label that node SOLVED
  - At each node, check which is the most promising path and mark it as part of the ***current best path.***


Before step 1
A (5)
Before step 2
A (6)
(9)
B (3)
C (4)
D (5)
Before step 3
A
(9)
B (3)
C (4)
D (10)
(10)
E (4)
F (4)
Before step 4
A
(12)
B (6)
C (4)
D (10)
G (5)
H (7)
(10)
E (4)
F (4)

AND-OR graphs cont ...

- Backtracking is required to update **current best path** and also to update **h' value.**
- Individual paths from node **A** to node **B** cannot be considered better or worse than any existing path to node **B** if any of the ancestors of node **B** are connected via AND arcs.

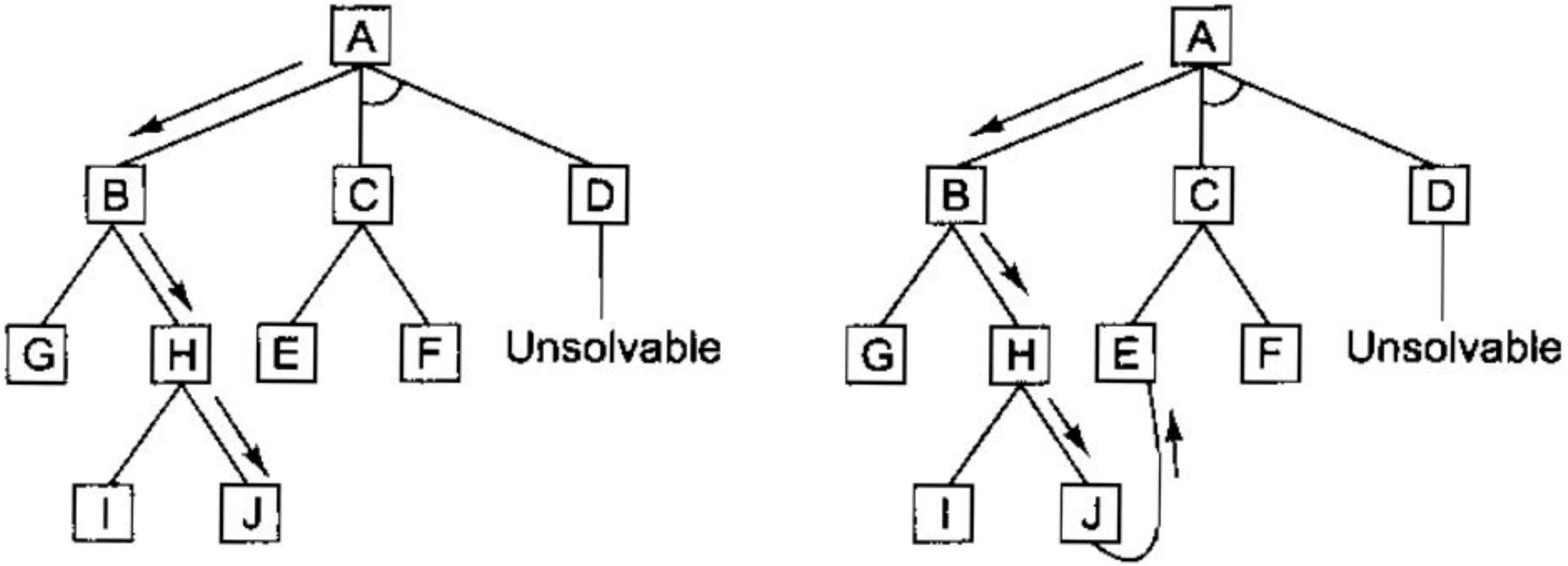
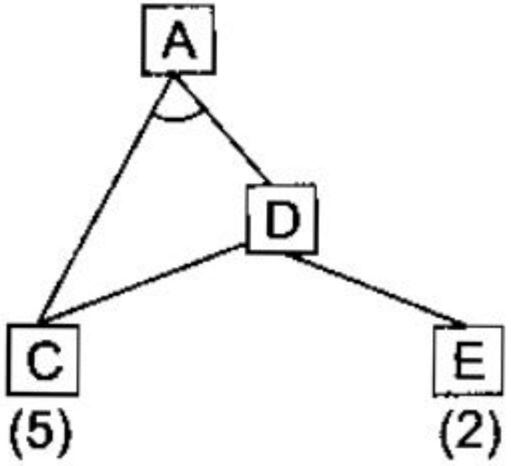


Fig. 3.9 *A Longer Poth May Be Better*

Limitation

- No interaction between subgoal nodes.



AO* algorithm

- The AND-OR graph algorithm is a simplification of the AO* algorithm
- First presented in *Martelli and Montanari [1973], Martelli and Montanari [1978] and Nilsson [1980].*
- Does not use OPEN or CLOSED lists, but uses a GRAPH structure that details the part of the graph that has been explicitly generated.
- Nodes point to its immediate predecessor(s)/parent(s) and to its immediate successor(s)/children
- Nodes also store **h'** (note: difference from A* where **g** is also stored)
- **Note:** Arcs present in the **current best path** are called ***labelled arcs.***

AO* Algorithm

- Initialize GRAPH with INIT (start state)
- Calculate **h’**(INIT)
- while(**h’(INIT) != SOLVEd or h’(INIT) != FUTILITY**)
    - Trace **current best path** to unexpanded nodes.
    - Select an unexpanded node, call it NODE
    - Generate children C[ ] of NODE
        - For each C in C[ ]
            - If C is not an *ancestor* of Node
                - Add C to GRAPH
                - If C is a terminal node
                    - **h’**(C) = 0 or SOLVED
                - Else
                    - Calculate **h’(C)**

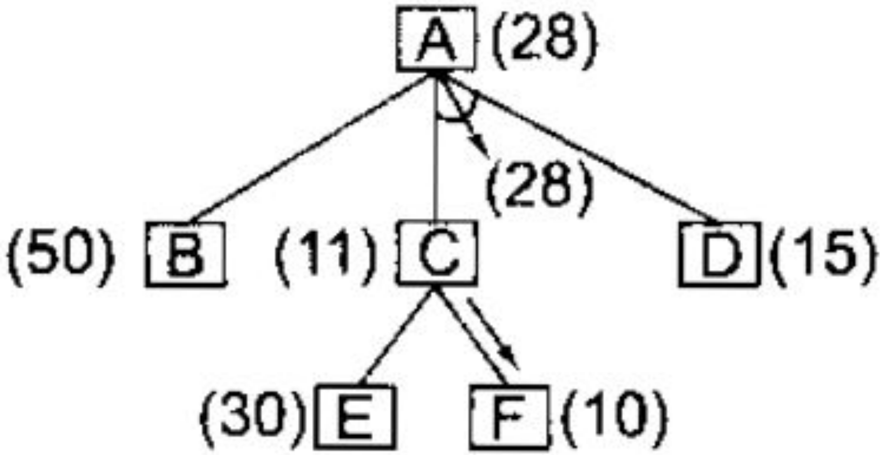
Cont ..

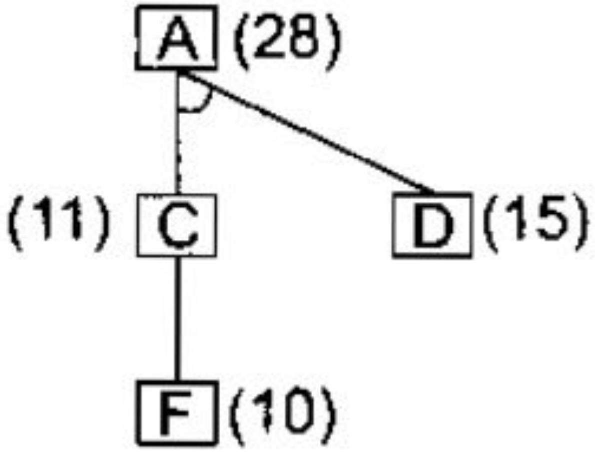
- Propagate changes back up graph
  - Initialize set S to contain NODE
  - While (!S.isEmpty)
    - CURRENT = Select node from S [prioritize nodes whose descendants are not in S]
    - Compute cost of all arcs generated from CURRENT
      - **h’** of all nodes at the end of arc + cost of arc itself
    - Set **h’** of CURRENT to minimum **h’** of arcs + arc cost
    - Mark arc with minimum **h’** as part of **current best path**
    - If all nodes from the arc included in **current best path** from CURRENT are SOLVED
      - **h’(CURRENT)** = **SOLVED**
    - If **h’**(CURRENT) has changed
      - Add all ***ancestors**** ***** of CURRENT to S.

NOTE ***: book says **ancestors** but **immediate parents** may be correct? Ancestors will work as well as SETs will remove duplicate members.

What if there are cycles?

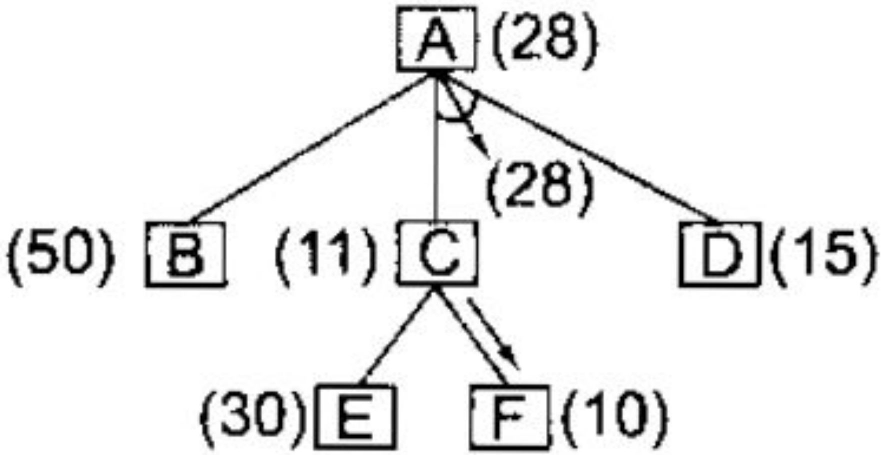
- How will backpropagation terminate??
- What if we expand F and the only child is A??

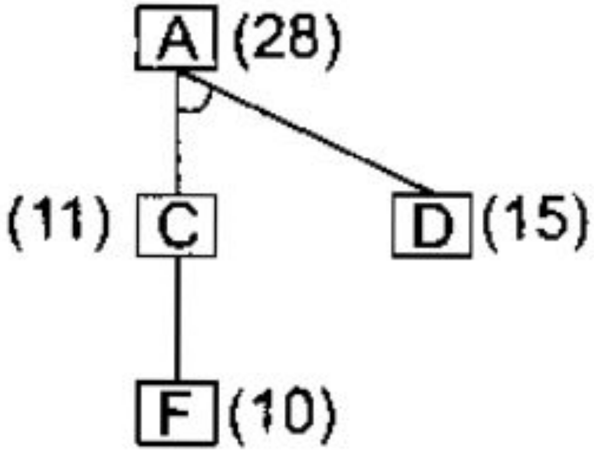




What if there are cycles?

- How will backpropagation terminate??
- What if we expand F and the only child is A??

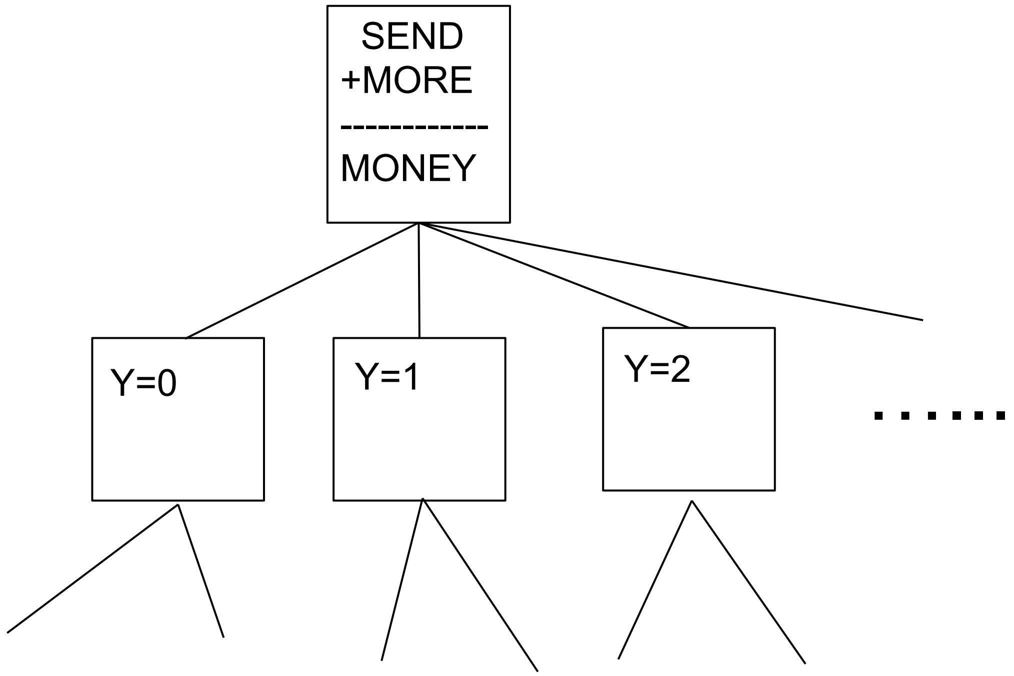




Set **h’(F)** to FUTILITY maybe??

Constraint Satisfaction

- Many AI problems can be viewed as constraint satisfaction problems where the goal is a state that satisfies a set of given constraints.
- Aims the reduce the amount of time spent searching for a solution by reducing the search space.
- Avoids making random guesses until necessary.


SEND
+MORE

MONEY
Y=0
Y=1
Y=2
......

Cont ..

- Constraint satisfaction operates in a space of constraint sets.
- Goal state is a state that has been constrained “enough” (depends on the problem)

Cont ..

- Consists of two main steps:
    - Propagate Constraints.
        - Through dependencies.
        - Through inferences.
            - $A = B + C$, $C = 3$
            - What can be propagated?
                - $A = B + 3$
    - Start Guessing (Searching)
        - This is done when propagation of constraints cannot proceed any further.
        - Let us extend the above example,
            - Let’s say we guess, $A = 5$
            - This will be added as a new constraint
            - And constraint propagation starts again
                - New constraint: $5 = B + 3$ or $B = 2$

Cont ..

- When does propagation of Constraints stop?
  - Detection of solution.
  - Detection of contradiction.
    - If the contradiction is with one of the propagated rules i.e., not the original set of constraints provided with the problem. Then we can say that no solution is consistent with all the current constraints
    - If the contradiction is with one of the original constraints, then the problem has no solution.
  - Cannot infer any more constraints
    - In which case guessing starts

Constraint Satisfaction Algorithm

- OPEN = Set of all objects that need to be assigned a value in the final solution.
- Loop until inconsistency is detected or when a solution is found
  - Select an Object (OB) from OPEN
  - Propagate (through dependencies or inference) the constraints associated with OB.
  - Create the list of Constraints associated with OB (NEW_LIST_OB).
  - Let the list of constrained created the last time OB was examined be OLD_LIST_OB (may be NULL)
  - If OLD_LIST_OB != NEW_LIST_OB
    - Set.add(all objects that share a constraint with OB)

Cont..

- If union of the constraints discovered in the previous steps gives us a solution
    - Quit and report solution
- If union of the constraints discovered in the previous steps gives us a contradiction
    - Return failure
- Else,
    - Loop until solution is found or no solution is possible
        - Select an object whose value has not been determined and find a way to strengthen the constraints associated with the object (make a ***guess*** at a value of that object or a constraint related to that object)
        - Recursively call Constraint satisfaction with the current set of constraints (including our guess)

Cont ..

- The algorithm is presented in a general format
- To apply to a actual AI problem, we require the use of two rules
    - Rules to propagate constraints for that specific problem
    - Rules to make appropriate guesses for that problem

Problem:

```
  SEND
+ MORE
......
 MONEY
```

Initial State:

No two letters have the same value.
The sums of the digits must be as shown in the problem.

Fig. 3.13 *A Cryptarithmetic Problem*

Steps

- Propagate constraints
    - Order is irrelevant
- Make Guesses
    - Order is important
        - Heuristics -> better to guess at variables that have less possible assignments than on variables that have more
        - Heuristics -> better to guess at variables that participate in more constraints
- LET C1, C2, C3 and C4 be the carry at tens, hundreds, thousands and ten thousands respectively

- M = 1, since two single-digit numbers plus a carry cannot total more than 19.
- S = 8 or 9, since S + M + C3 > 9 (to generate the carry) and M = 1, S + 1 + C3 > 9, so S + C3 > 8 and C3 is at most 1.
- O = 0, since S + M(l) + C3 (<= 1) must be at least 10 to generate a carry and it can be at most 11. But M is already 1, so O must be 0.
- N = E or E + 1, depending on the value of C2. But N cannot have the same value as E. So N = E + 1 and C2 is 1.

- In order for C2 to be 1, the sum of N + R + Cl 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.

No more Constraints can be propagated, so we guess

- E = 2

- N = 3, since N = E + 1.
- R = 8 or 9, since R + N (3) + C1 (1 or 0) = 2 or 12. But since N is already 3, the sum of these nonnegative numbers cannot be less than 3. Thus R + 3 + (0 or 1) = 12 and R = 8 or 9.
- 2 + D = Y or 2 + D = 10 + Y, from the sum in the rightmost column.

Dependency Directed Backtracking: Mentioned in pg 72. E.Rich & K.Knight

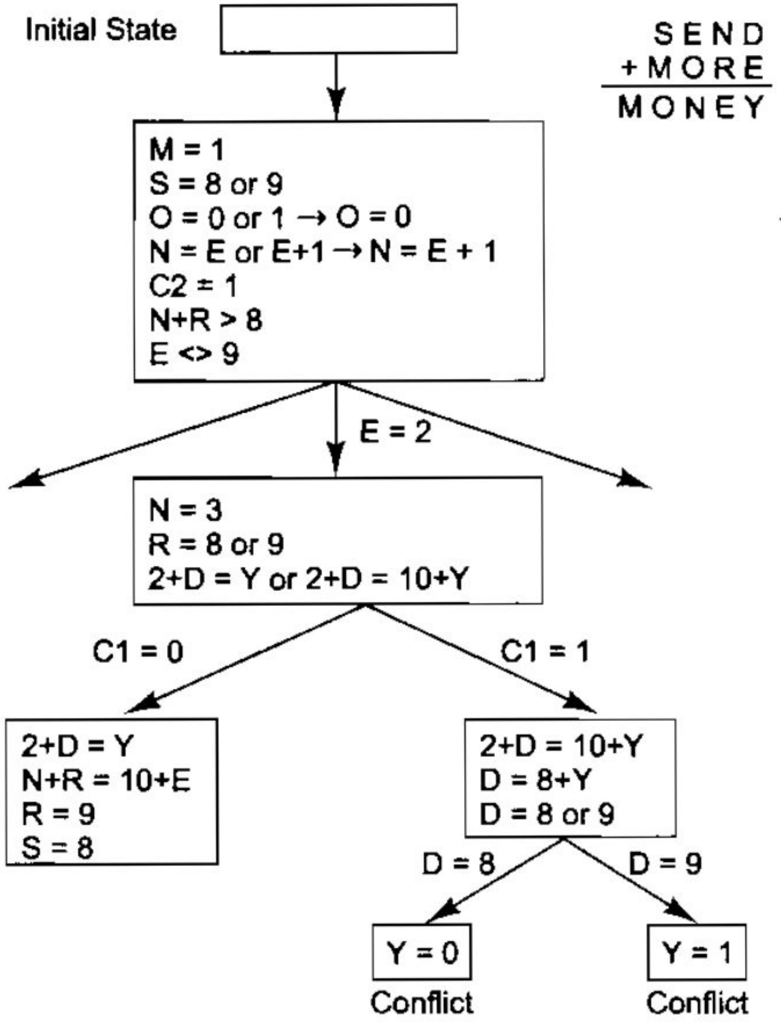


Fig. 3.14 *Solving a Cryptarithmetic Problem*

Means-Ends Analysis

1. Based on ***differences*** between current state and goal state
2. Finds an operator/action that can reduce the difference
3. If operator cannot be applied to current state
   a. Define a new state to which operator can be applied to and call it a ***subgoal***.
   b. Repeat from step one with
      i. goal state = subgoal
4. If Operator does not produce the final goal (**G**) but creates a state **S**
   a. Set current state = S, goal state = G
   b. Repeat from step 1.

Cont ..

- Rules
  - Does not have complete state information for both left (preconditions) and right hand side.
  - Aspects that change in the current state after application of action
- Difference table
  - Indexes rules by the differences they can be used to reduce.

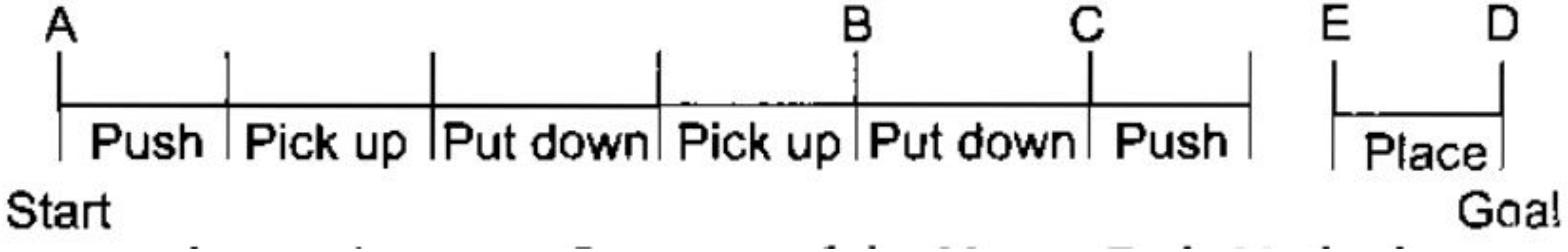
Operator	*Preconditions*	*Results*
PUSH(obj, loc)	at(robot, obj)^ large(obj)^ clear(obj)^ armempty	at(obj, loc)^ at(robot, loc)
CARRY(obj, loc)	at(robot, obj)^ small(obj)	at(obj, loc)^ at(robot, loc)
WALK(loc)	none	at(robot, loc)
PICKUP(obj)	at(robot, obj)	holding(obj)
PUTDOWN(obj)	holding(obj)	¬holding(obj)
PLACE(obj1, obj2)	at(robot, obj2)^ holding(obj1)	on(obj1, obj2)

Fig. 3.15 *The Robot's Operators*

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

Fig. 3.16 *A Difference Table*

- Move Table with two objects on it from one location to another.
- Most significant difference is the location of the table.
- To move table, we can either CARRY or PUSH
- CARRY has precondition -> robot has to be at obj ^ obj has to be small
  - Since current state robot is not at obj, therefore
    - Create new subgoal, where subgoal = robot is at obj
    - Can use WALK operator to move robot to be positioned next to obj
  - Check next precondition
    - .....



Algorithm: Means-Ends Analysis (CURRENT, GOAL)

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