ECE 479/579
Principles of Artificial Intelligence – Part III
Spring 2005

Dr. Michael Marefat (marefat@ece.arizona.edu)

Initial State:

Ontable(B)

Ontable(A)

On (C,A)

Clear(C)

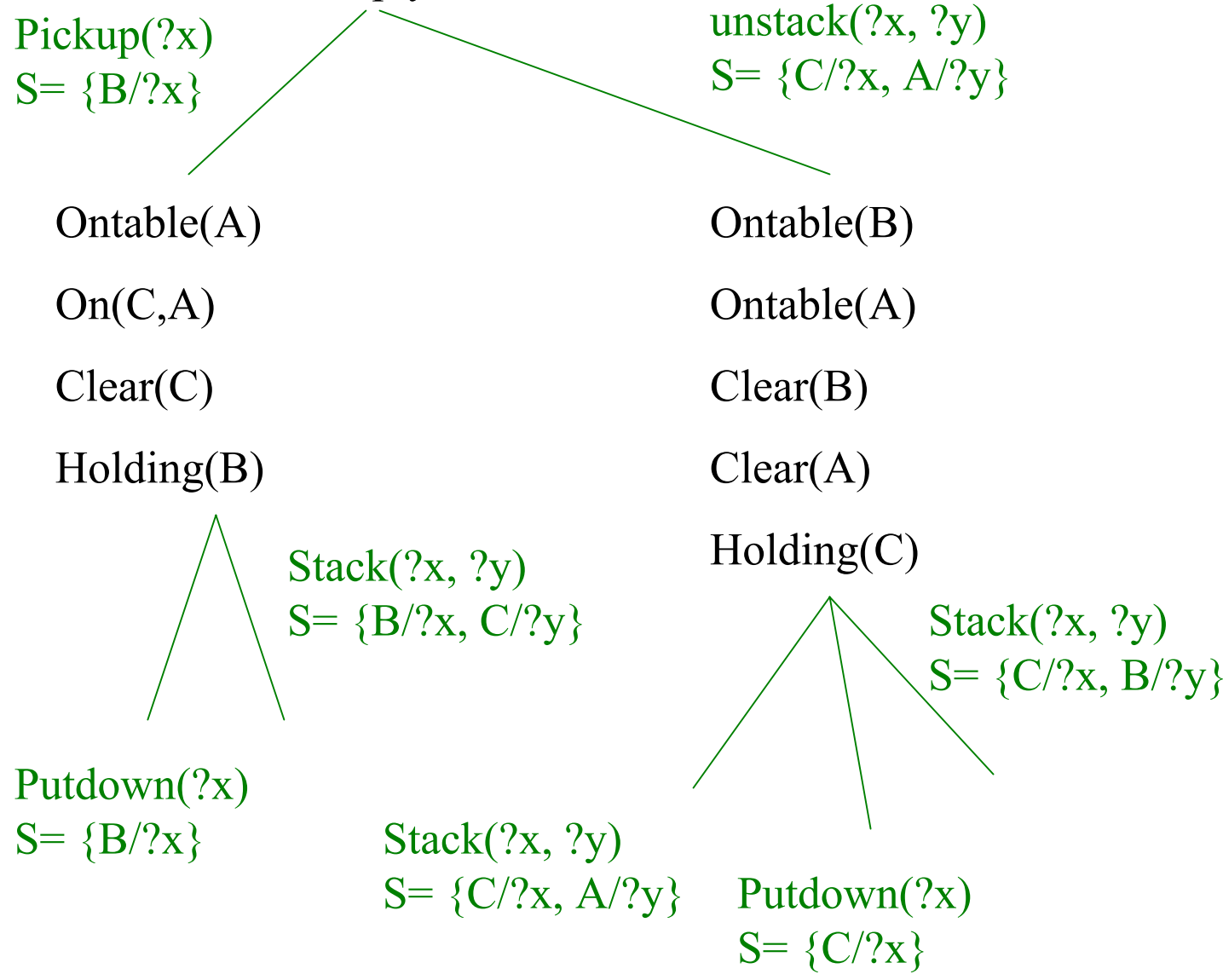
Clear(B)

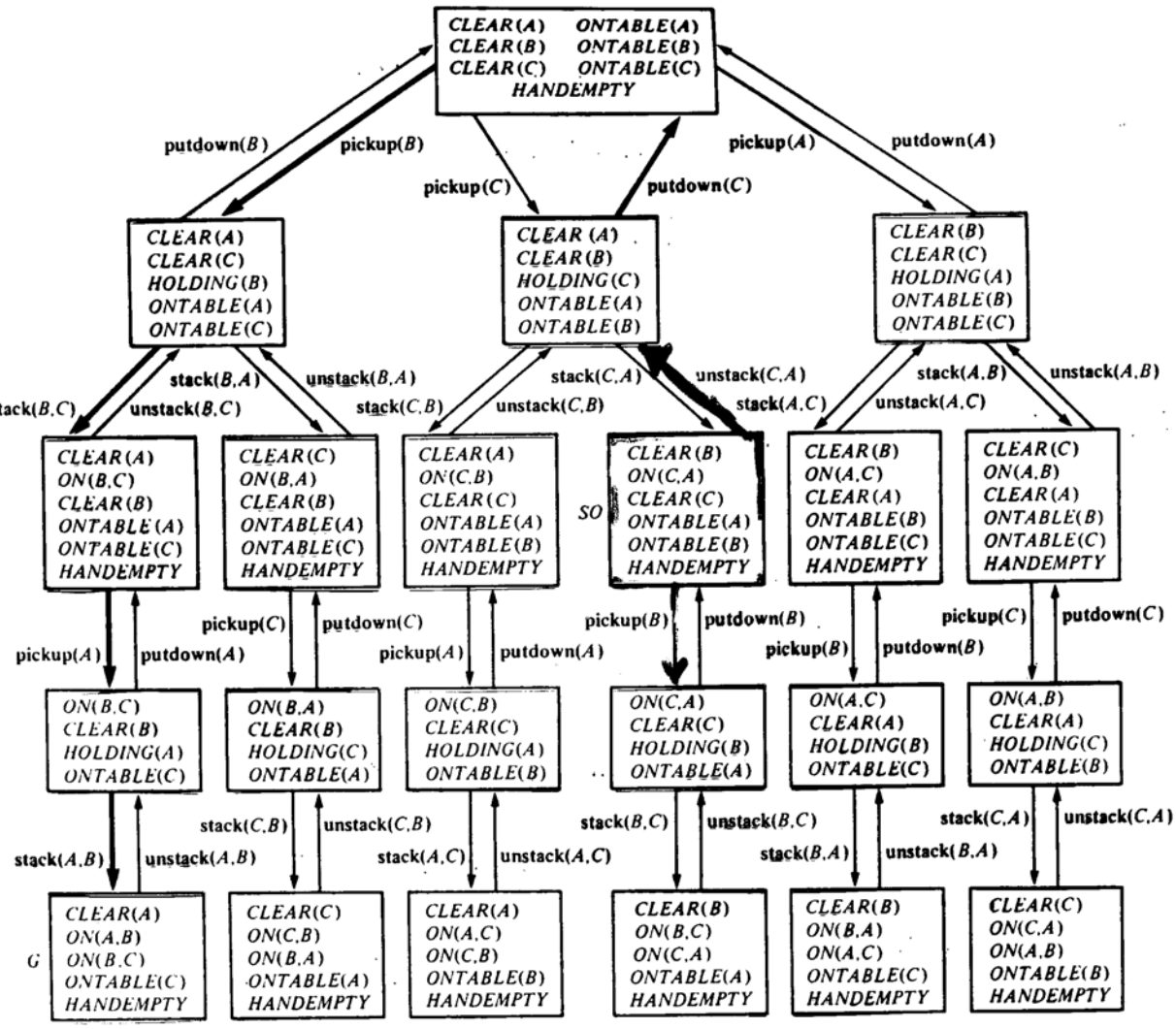
Handempty

Matching actions:

Pickup(?x) {B/?x}

unstack(?x,?y) {C/?x, A/?y}

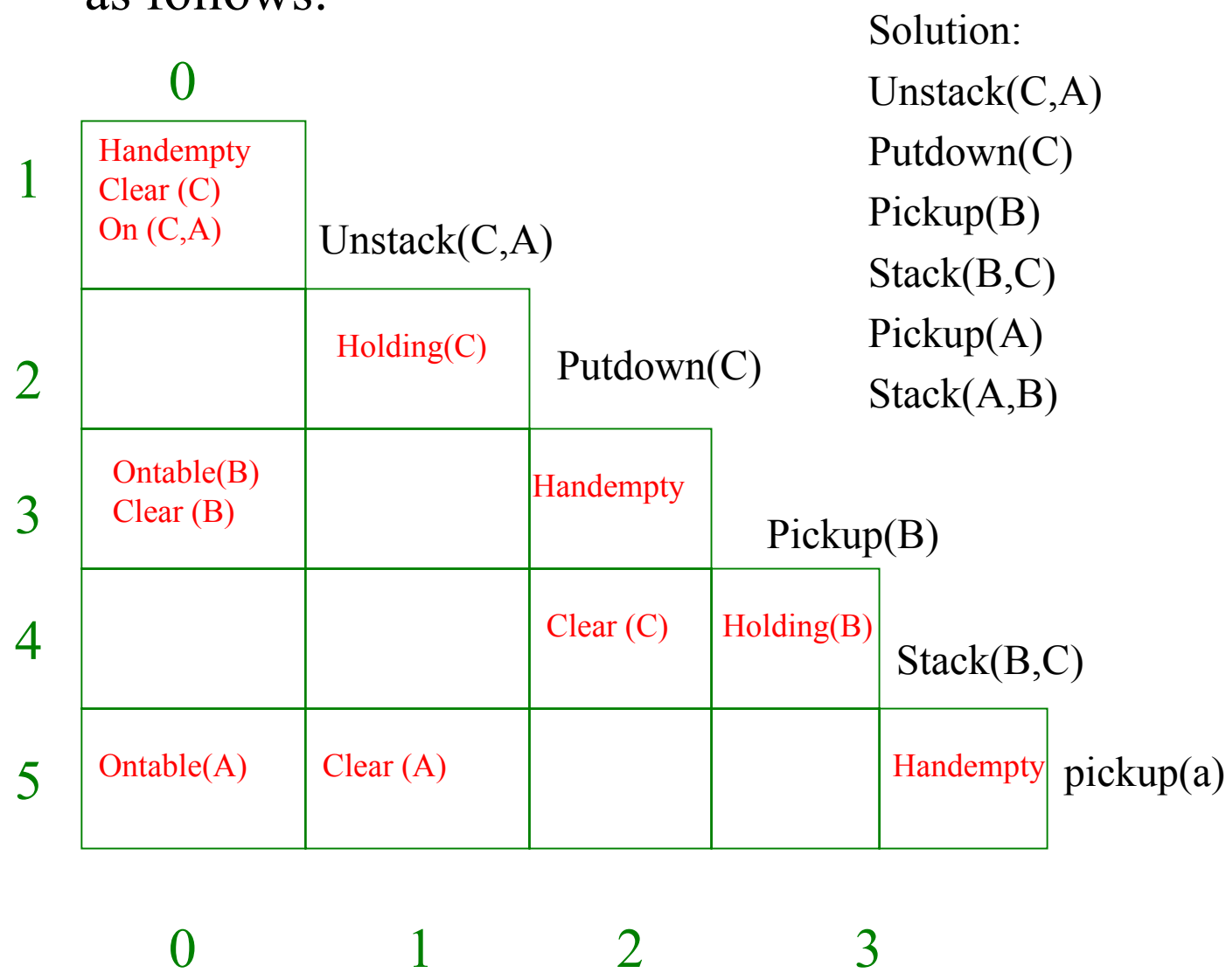




The state-space for a robot problem.

Triangle method for Execution and recovery from Errors:

Once we have generated the solution, then construct a triangle with the steps in the solution as follows:



Backward Planning / Problem Solving:

We have a goal G = G1∧G2 ∧…….. ∧GN

which we want to achieve, and we have initial conditions I= I1 ∧I2 ……..Im, we will consider subgoals G1, G2,…., and match actions which can produce them, then we regress all subgoals through the action. The regression plus the preconditions of the action give us new subgoals G1' ∧ G2',…….

Continue this process until all subgoals match with initial conditions.

Regression

We have a goal:

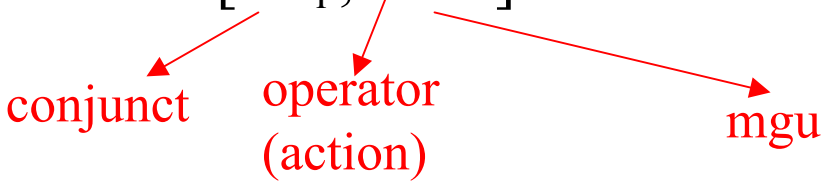
$$G = G1 \wedge G2 \wedge \ldots\ldots\ldots GN$$

we have an operator that can unify with L (there is a predicate in its add list that unifies with L). This predicate in add list of operator can be denoted L':

$$L'S = L$$

mgu substitution

To regress G through this operator, take every conjunct other than L and compute the regression R[G_i , F s]



G_i'= R[G_i, Fs] =

T : if Fs has literal G_i in its add list.

F : if Fs has literal G_i in its delete list.

G_i s : if neither of the above.

This works if Fs is a ground action.

$\Rightarrow$ no uninstantiated variables remaining.

Ex: Consider the goal

G = Holding(B) ∧ Handempty ∧ Ontable (C) ∧ Clear (C)

L = Holding (B)

F = Unstack(?x, ?y) S= { B/ ?x , C/ ?y }

Precondition list of F:

Handempty ∧ Clear (?x) ∧ On(?x, ?y)

Delete list of F:

Handempty ∧ Clear (?x) ∧ On(?x, ?y)

Addlist of F:

Holding(?x) ∧ Clear (?y)

R[Handempty, unstack(?x, ?y) {B/?x, C/?y}] = **F**

R[Ontable(C), unstack(?x, ?y) {B/?x, C/?y}] =

Ontable(C)

R[Clear (C), unstack(?x, ?y) {B/?x, C/?y}] = **T**

Regressing when we have uninstantiated variables

If a variable does not have an assigned constant in S, then regression of Gi containing this variable could result in different possible results.

Ex:

G = Holding(B) $\wedge$ Handempty $\wedge$

Ontable(C) $\wedge$ Clear (C)

L = Holding (B)

F = unstack(?x, ?y) S = {B/?x}

R[Handempty, FS] = F

R[Ontable(C), FS] = Ontable (C)

R[Clear(C), FS] = ?y = C $\rightarrow$T

?y != C $\rightarrow$ Clear(C)

Whenever more than one regression is possible for G_i, $G_i^{1'}$ or $G_i^{2'}$, then we generate two children states, one for each possible value of the variable.

the 2nd outcome for the above regression is listed as

neq (?y C) $\wedge$ Clear (C)

The same method applies when G_i is in the delete list.

Regressing when we have uninstantiated variables in the delete list:

Similar to the case for the add list:

Example:

G= Clear (C)

F = Unstack(?x, ?y) S= { B/ ?y}

R[Clear(C), FS] = { ?x=C → F ; ?x != C → Clear (C) }

logically one can write:

[equal(?x, C) ⇒F] ∧ [neq (?x, C) ⇒ Clear (C)]

Thus the regression generates the following subgoal:

[neq(?x, C) ∧ Clear(C)]

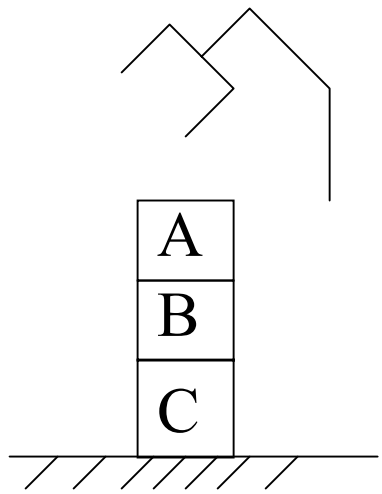
Summary:

The goal expression has a literal that unifies with one of the literals in the add list. The subgoal expression is created by:

1. Regress the non-matched literals of the goal through the action/ operator.

2. Generate the list that results from applying the substitution to the precondition list of the action/ operator.

3. Conjoin the results of 1 and 2.



Ex:

$G = On(A,B) \wedge On(B,C)$

$F = Stack(?x, ?y)$ $S = \{A/?x, B/?y\}$

subgoal:

$L = On(A,B)$

1. $R[On(B,C), Fs] = On(B, C)$

2. Precondition. $S = Holding(A) \wedge Clear(B)$

3. subgoal= $On(B,C) \wedge Holding(A) \wedge Clear(B)$

Algorithm for backward planning

STRIPS(G) G is the goal expression
S is the initial condition/ State

1. Until S matches G do

begin

2. g ← a component of G that does not match S; non-deterministic selection.
we can backtrack and make a different selection.

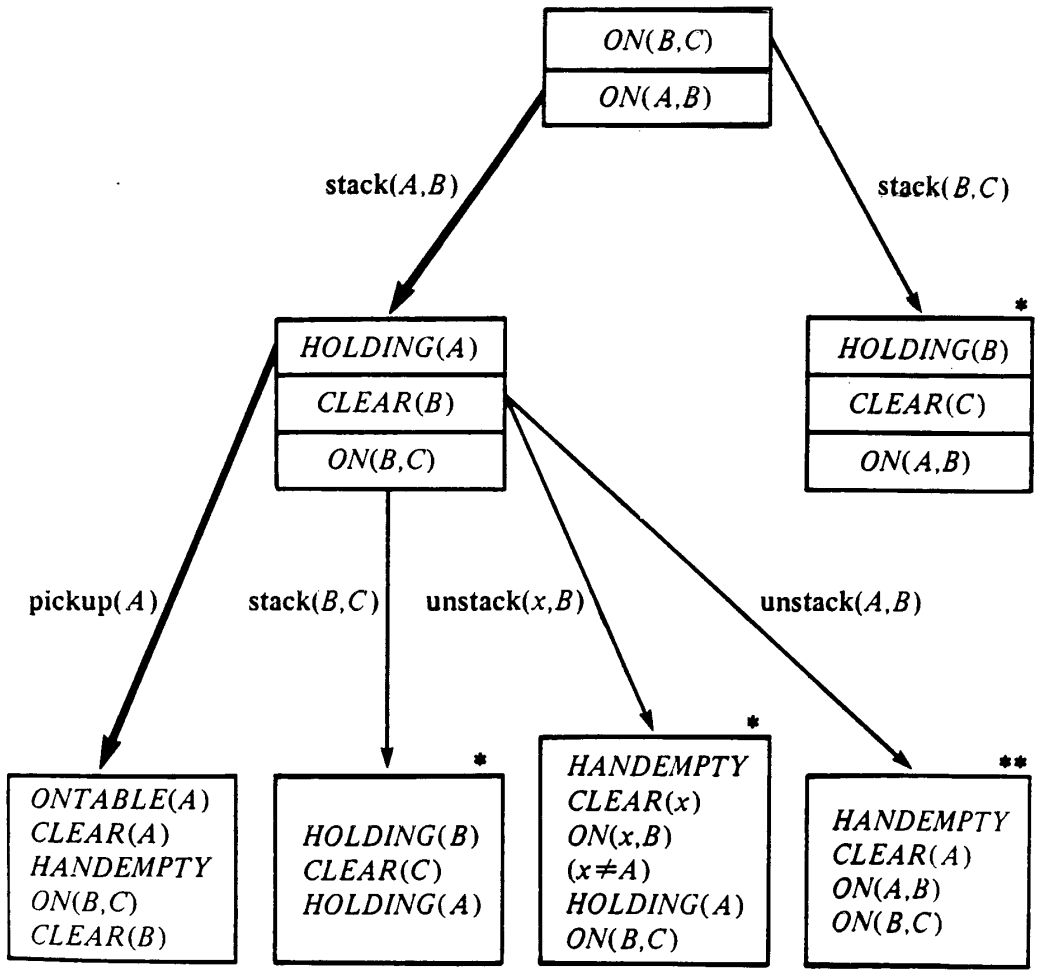
3. f ← an action/ operator whose add list contains a literal that matches g.
another point we backtrack to, if earlier selection no good.

4. p ← precondition formula for f with the unifying substitution applied to it.

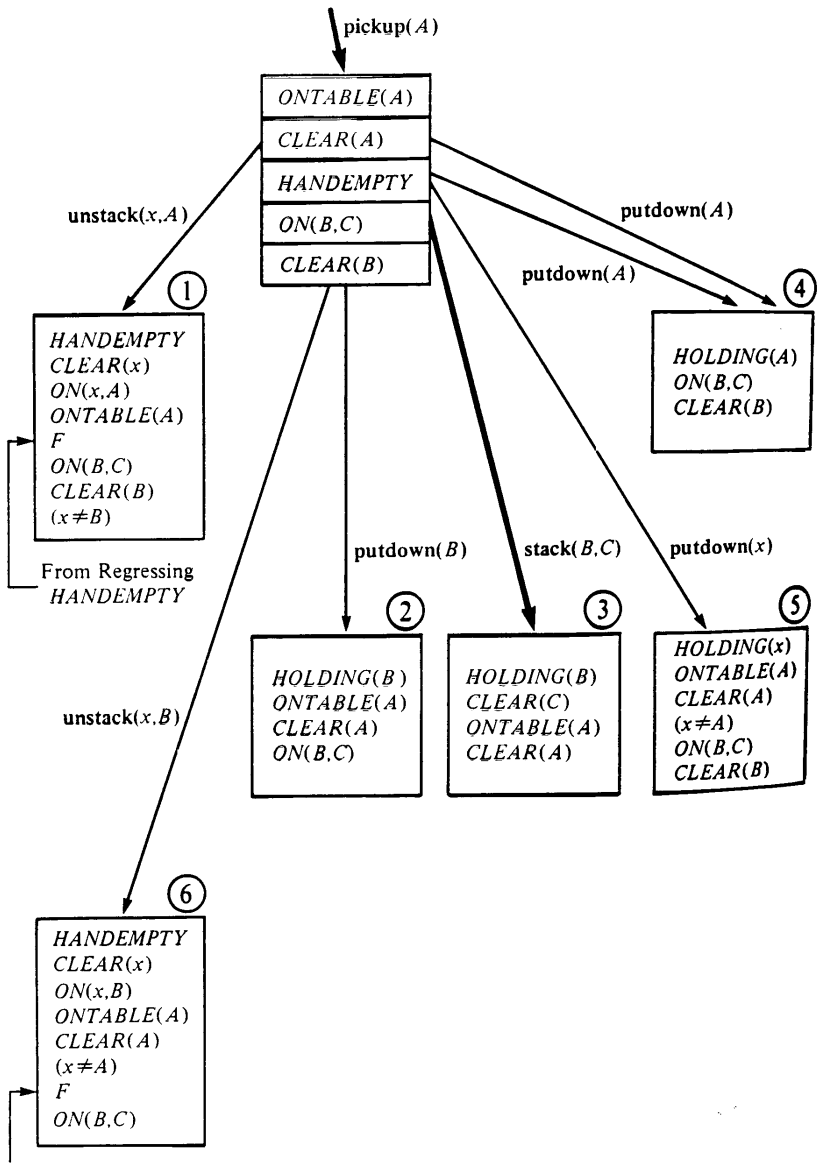
5. STRIPS(p) recursive call to solve the subproblem.

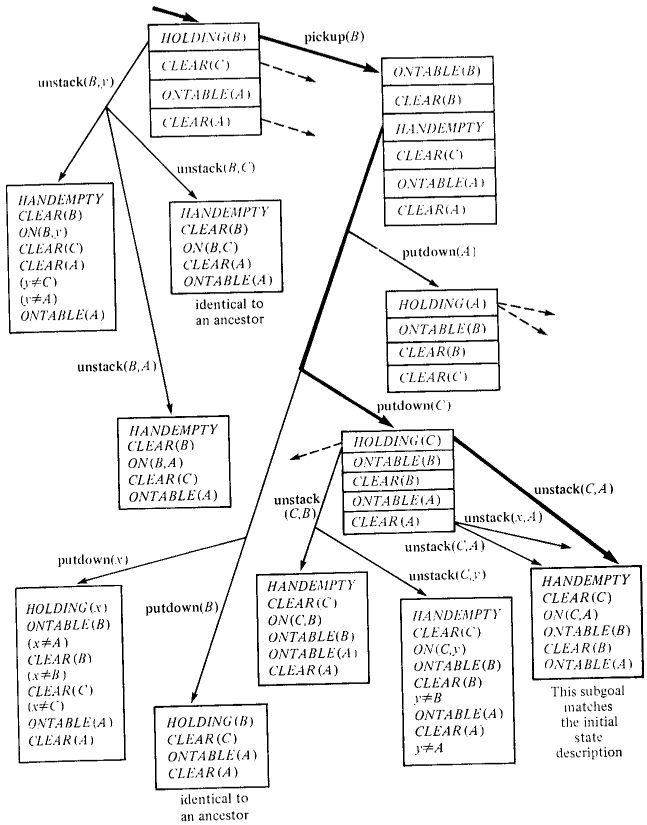
6. S ← result of applying the action f with the unifying substitution to S.

end

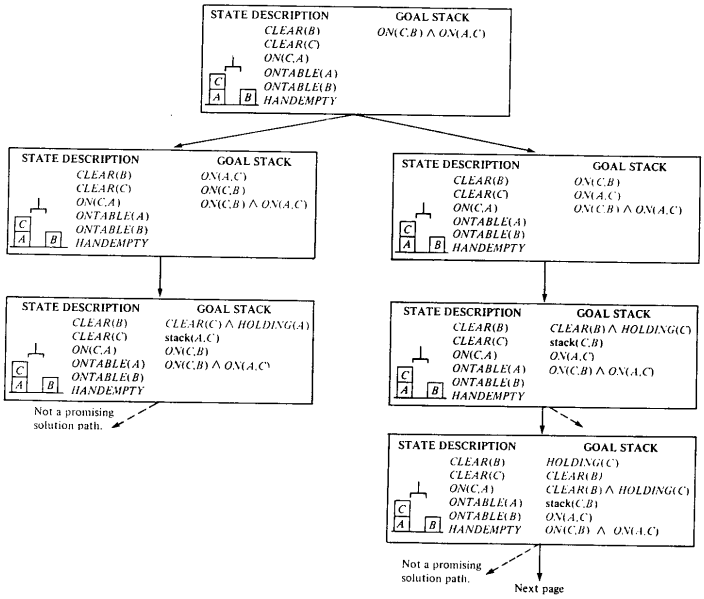


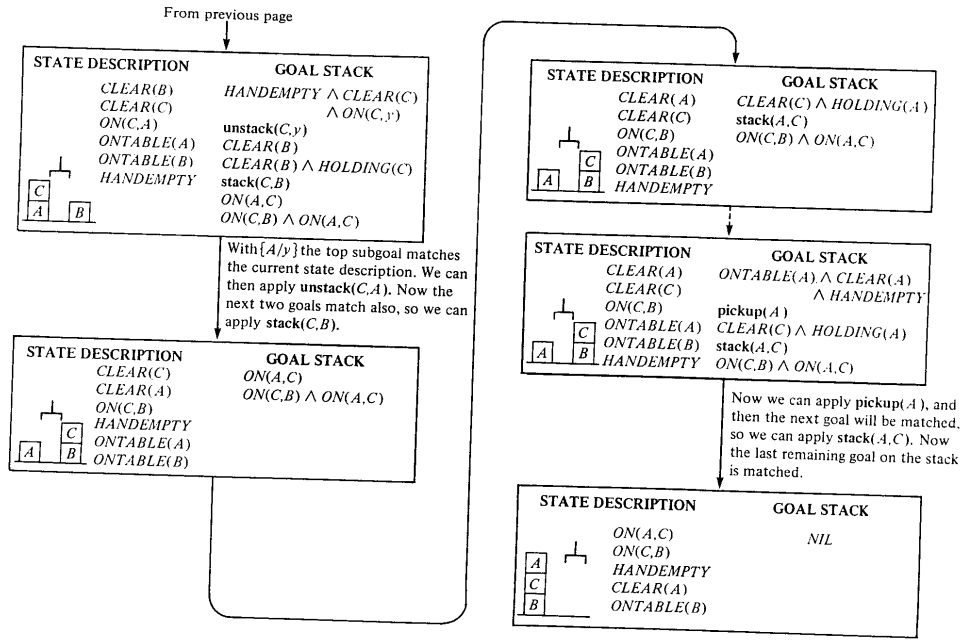
Part of the backward (goal) search graph for a robot problem.


pickup(A)
ONTABLE(A)
CLEAR(A)
HANDEMPTY
ON(B,C)
CLEAR(B)
unstack(x,A)
putdown(A)
putdown(A)
1
HANDEMPTY
CLEAR(x)
ON(x,A)
ONTABLE(A)
F
ON(B,C)
CLEAR(B)
(x≠B)
From Regressing
HANDEMPTY
4
HOLDING(A)
ON(B,C)
CLEAR(B)
putdown(B)
stack(B,C)
putdown(x)
2
HOLDING(B)
ONTABLE(A)
CLEAR(A)
ON(B,C)
3
HOLDING(B)
CLEAR(C)
ONTABLE(A)
CLEAR(A)
5
HOLDING(x)
ONTABLE(A)
CLEAR(A)
(x≠A)
ON(B,C)
CLEAR(B)
unstack(x,B)
6
HANDEMPTY
CLEAR(x)
ON(x,B)
ONTABLE(A)
CLEAR(A)
(x≠A)
F
ON(B,C)



Conclusion of the backward search graph.

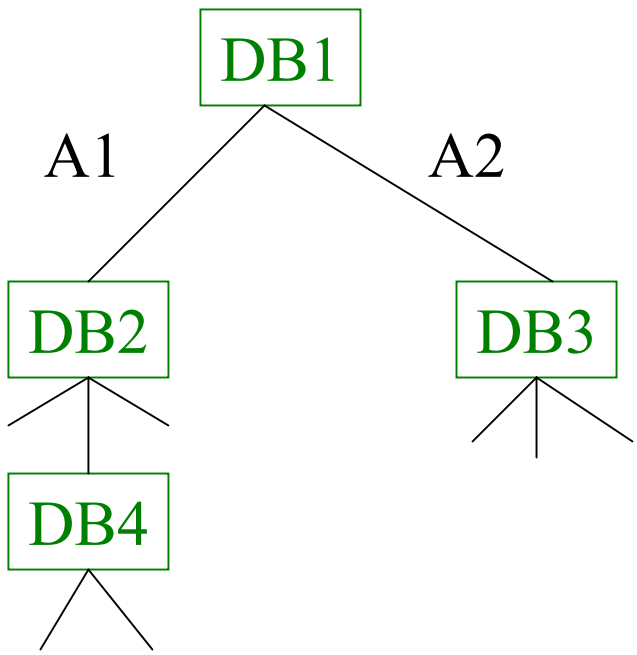

STATE DESCRIPTION
GOAL STACK
CLEAR(B)
CLEAR(C)
ON(C,A)
ONTABLE(A)
ONTABLE(B)
HANDEMPTY
ON(C,B) ∧ ON(A,C)
ON(A,C)
ON(C,B)
CLEAR(C) ∧ HOLDING(A)
stack(A,C)
HOLDING(C)
CLEAR(B)
CLEAR(B) ∧ HOLDING(C)
stack(C,B)
Not a promising solution path.
Next page


From previous page
STATE DESCRIPTION
GOAL STACK
HANDEMPTY ∧ CLEAR(C) ∧ ON(C,y)
unstack(C,y)
CLEAR(B)
CLEAR(B) ∧ HOLDING(C)
stack(C,B)
ON(A,C)
ON(C,B) ∧ ON(A,C)
With {A/y} the top subgoal matches the current state description. We can then apply unstack(C,A). Now the next two goals match also, so we can apply stack(C,B).
CLEAR(A)
CLEAR(C)
ON(C,B)
HANDEMPTY
ONTABLE(A)
ONTABLE(B)
CLEAR(C) ∧ HOLDING(A)
stack(A,C)
ONTABLE(A) ∧ CLEAR(A) ∧ HANDEMPTY
pickup(A)
CLEAR(C) ∧ HOLDING(A)
Now we can apply pickup(A), and then the next goal will be matched, so we can apply stack(A,C). Now the last remaining goal on the stack is matched.
NIL

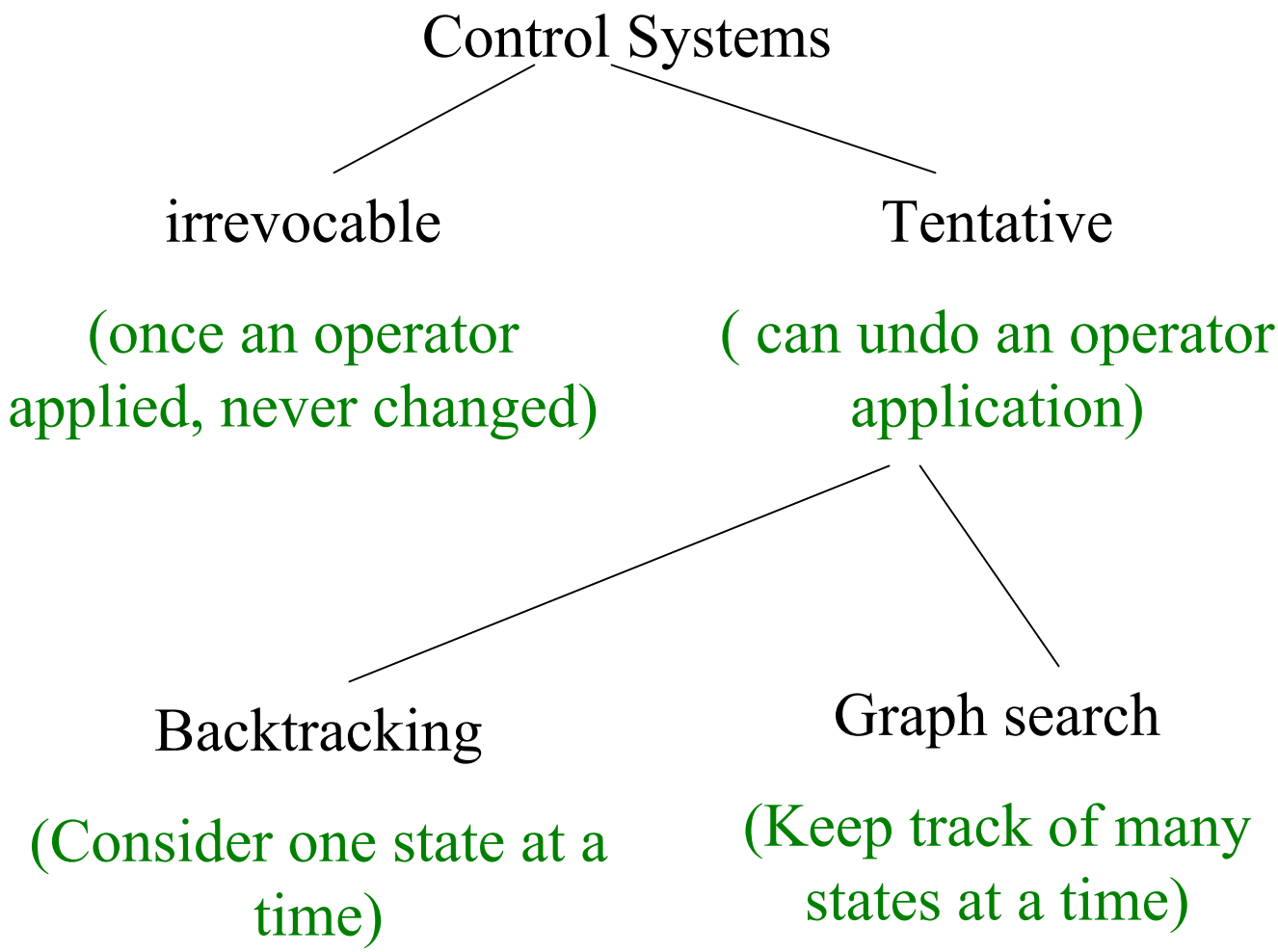
Production systems

Generally characterized by three elements:

1. A global database
2. A set of rules or actions/operators.
3. A Control system.



Control system chooses which operator to apply and keeps on applying operators until a terminal condition is reached.



1. Irrevocable control

1. Consider the current state of global database.

2. Apply an operator.

3. Back to step 1 until terminal condition is met.

Selecting the operator to apply:

usually done by a function evaluation. f(x)

Operator

Pick the operator for which f(x) is minimum.

Consider the problem of solving the 8 puzzle:

2	5	3
1	6	4
7		5

Initial

1	2	3
8		4
7	6	5

Goal

Set of operators

- blank up
- blank down
- blank left
- blank right

Define f(x) = - (No of tiles out of place)

f(initial) = - 4

f(blank left) = -5

f(blank right) = -5

f(blank up) = -3 ← Operator selected to apply

new DB

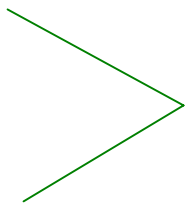
2	8	3
1		4
7	6	5

f(blank left) = -3

f(blank right) = -4

f(blank up) = -3

f(blank down) = -4



select one of the min f operators

Advantages of irrevocable:

- simple
- fast
- less memory

Disadvantages of irrevocable:

- can get stuck on local optima
- information about past selections lost

Backtracking control

try operators in a fixed order, and backtrack to a previous state when appropriate.

It is appropriate to backtrack when we come up with a state we have already seen before.

Ex: Consider the 8 puzzle example:

Fixed order of operators can be

blank left

blank right

blank up

blank down

Algorithm for Backtracking

Backtrack(DATA)

1. If TERM(DATA), return nil

Test whether terminal condition is satisfied.

2. If DEADEND(DATA), return fail.

All possible operators/actions are tried and exhausted.

3. Applicable-Operators ← Operators(DATA)

4. Loop:

   4.1 If NULL(Applicable_Operators), return fail

   no operator can be applied.

   4.2 R ← first(Applicable_Operators)

   Applicable_Operators ← rest (Applicable_Operators)

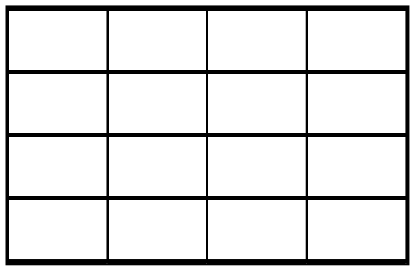
   4.3 newDATA ← R (DATA)

   4.4 Path ← Backtrack(newDATA)

   4.5 If Path= fail goto Loop

   4.6 return Cons(R, Path)

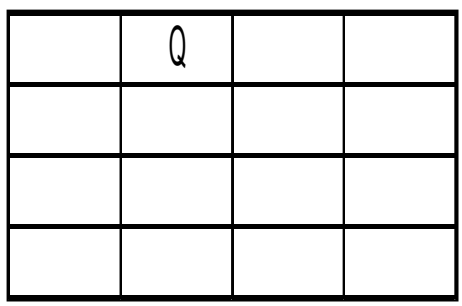
DATA=



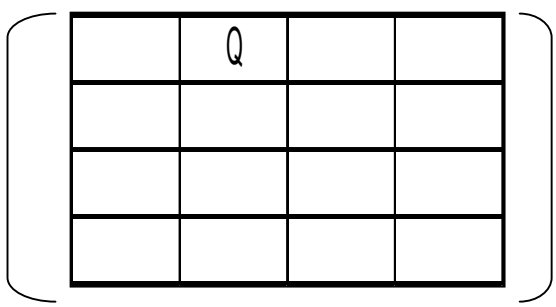
Any Row's possibilities:

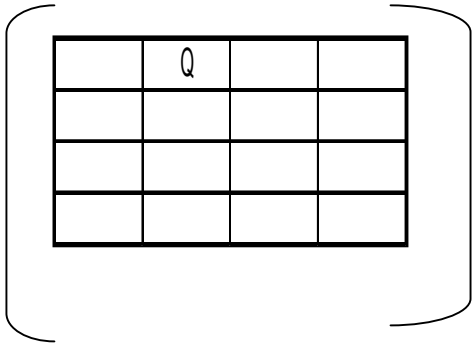
$R_{i1}, R_{i2}, R_{i3}, R_{i4}$

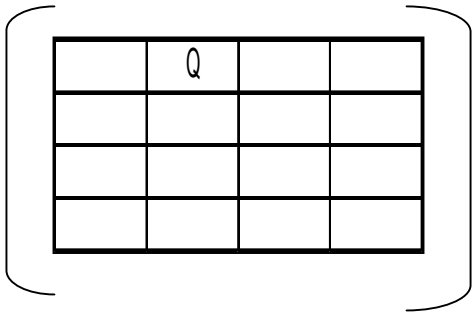
1. TERM(DATA) X
2. DEADEND(DATA) X
3. Rules = $(R_{12}, R_{13}, R_{11}, R_{14})$
4. Null(Rules) X
5. $R \leftarrow R_{12}$
6. Rules $\leftarrow (R_{13}, R_{11}, R_{14})$
7. RDATA $\leftarrow$



8. Path $\leftarrow$ Backtrack



1. Term  ? X

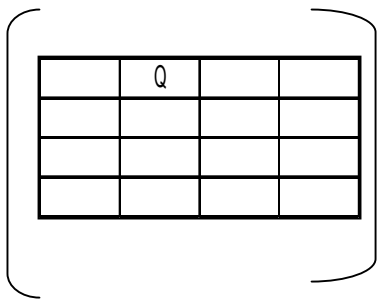
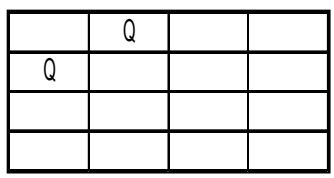
2. DEADEND  ? X

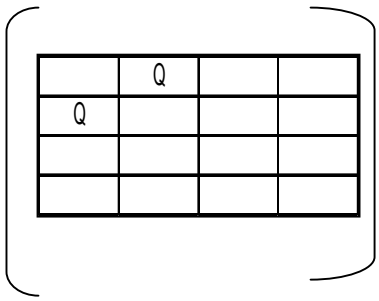
3. Rules ← (R_{21},R_{24},R_{22} ,R_{23})

4. Loop: Null (Rules)

5. R ← R_{21}

6. Rules ← (R_{24},R_{22} ,R_{23})

7. RDATA ← R_{21}  = 

8. Path ← Backtrack 

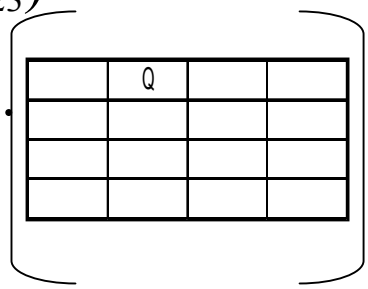
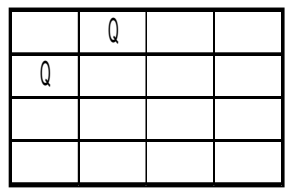
Path ← fail

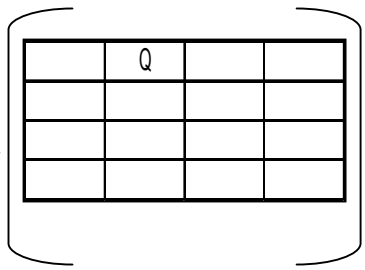
9. Path = fail ⇒ goto Loop

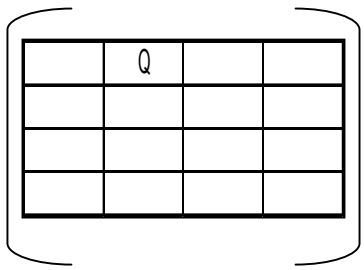
4. Null(Rules) = NULL(R_{24},R_{22} ,R_{23})? X

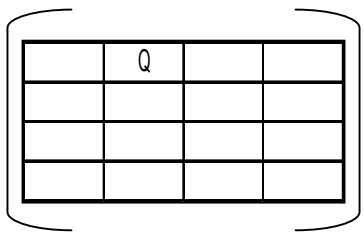
5. R← R_{24} .

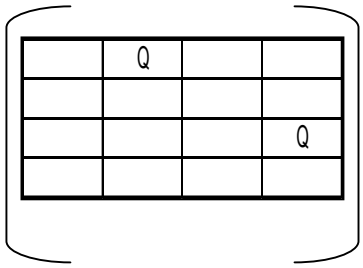
6. Rules ← (R_{22} ,R_{23})

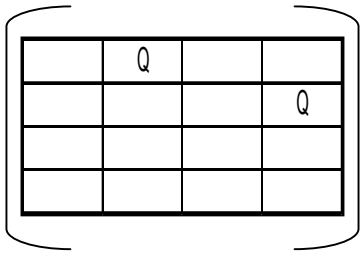
7. RDATA ← R_{24} .  = 

8. Path ← Backtrack 

1. TERM  ? X

2. DEADEND  ? Yes → Return Fail

1. TERM  ? X

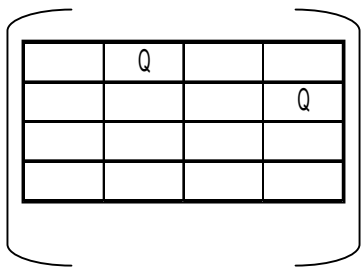
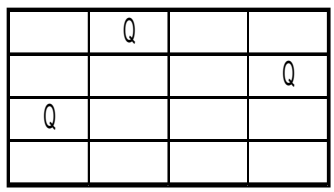
2. DEADEND  ? X

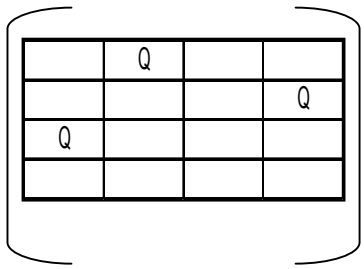
3. Rules $\leftarrow (R_{31}, R_{34}, R_{32}, R_{33})$

4. Loop: NULL(Rules) X

5. R $\leftarrow R_{31}$.

6. Rules $\leftarrow (R_{34}, R_{32}, R_{33})$

7. RDATA $\leftarrow R_{31}$  = 

8. Path $\leftarrow$ Backtrack 

Control Systems using Graph search

we keep track of all previous states at every iteration of the algorithm. We pick the most suitable state and expand from it.

Definitions about graphs

- A graph consists of sets of nodes and arcs which connect the nodes

- If an arc from node n_i to node n_j, then n_i is the parent node of n_j, and n_j is the child or successor of n_i.

-If no node in the graph has more than one parent, then we have a tree.

- A node without a parent is the root.

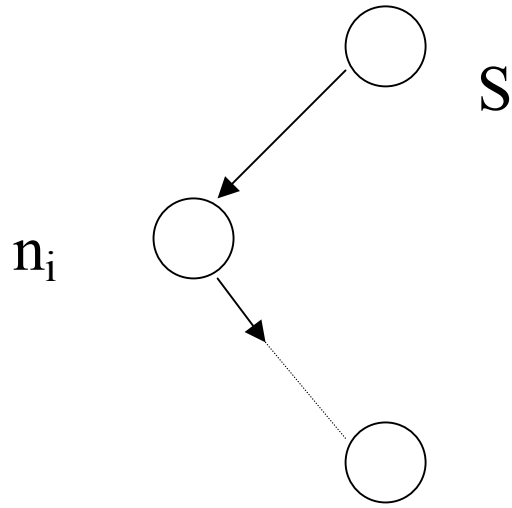
- A terminal node is a node without a child.

- A sequence of nodes

$(n_{i0}, n_{i2}, \ldots\ldots, n_{ik})$ with each n_{ij} a child of

n_{ij-1} is a path of length K from n_{i1} to n_{ik}.

- C (n_i, n_j) is the cost of the arc from n_i to n_j.



We may have more than one goal node, i.e. set of [t_i]

t goal node.

Ordering of nodes in graph Search

1. Depth first Search:

Order nodes in the open list in descending order of depth from S.

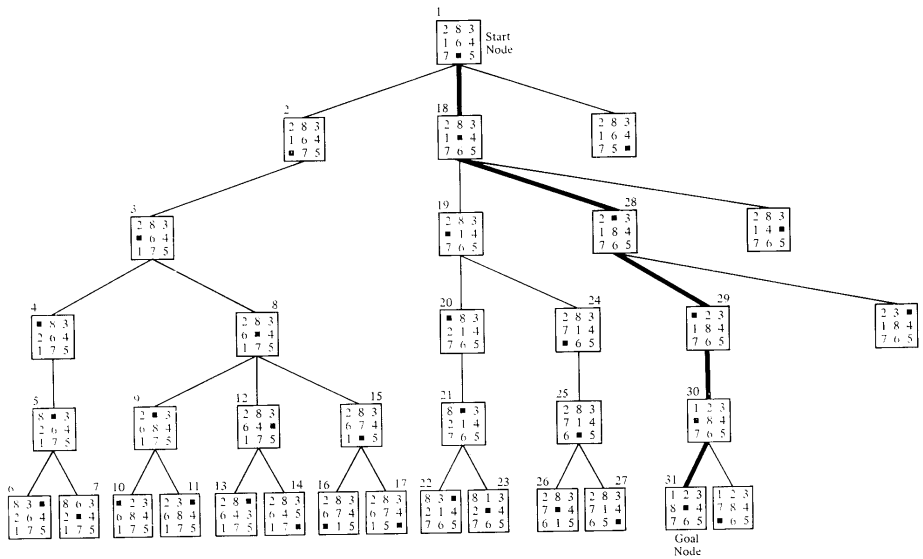
2. Breadth first search:

Order nodes in the open list in increasing order of depth from S.

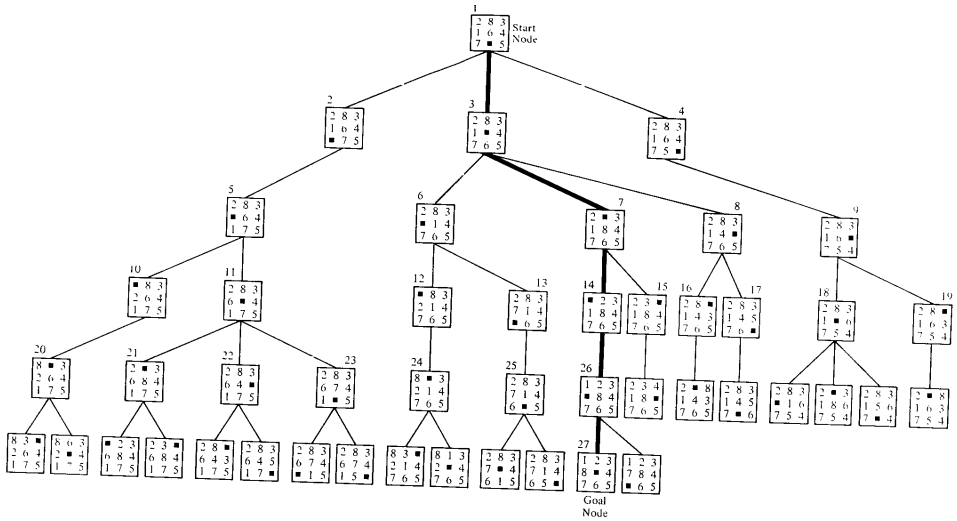
Breadth first will find the shortest length path to goal.

Both breadth_first and depth_first are un_informed.

Heuristic search uses problem information.



A search tree produced by a depth-first search.



A search tree produced by a breadth-first search.

<<< Refer to Section 4.1 of the text >>>

Heuristic Search

Uses problem information to reduce search space.

Use an evaluation function, f( ) to measure the promise of a node.

for every node n.

f(n) - value of f for goodness of node.

Ex: Consider 8-tile puzzle,

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

g(n)= depth of node n from s.

h(n)= no. of tiles out of place.

Ex2: Manhattan distance:

Consider the 8-tile puzzle.

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

g(n) = depth of node n from s.

$h(n) = \sum_i$ block distance cell I is out of place from its desired position.

A GENERAL GRAPH-SEARCHING PROCEDURE

The process of explicitly generating part of an implicitly defined graph can be informally defined as follows.

Procedure GRAPHSEARCH

1 Create a search graph, G, consisting solely of the start node, s. Put s on a list called OPEN.

2 Create a list called CLOSED that is initially empty.

3 LOOP: if OPEN is empty, exit with failure.

4 Select the first node on OPEN, remove it from OPEN, and put it in CLOSED. Call this node n.

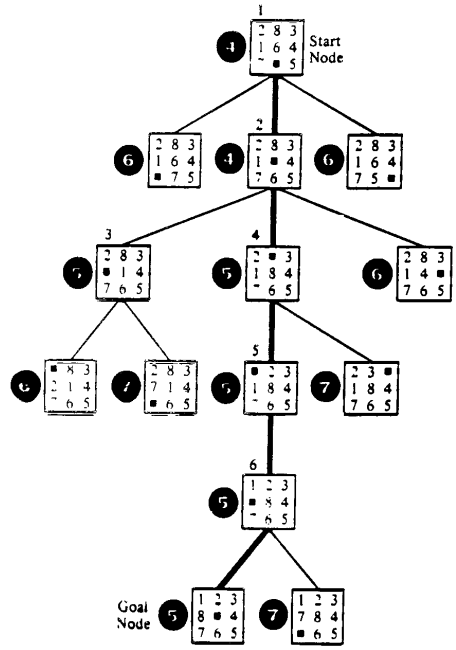
5 If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G. (Pointers are established in step 7.)

6 Expand node n, generating the set, M, of it's successors and install them as successors of n in G.

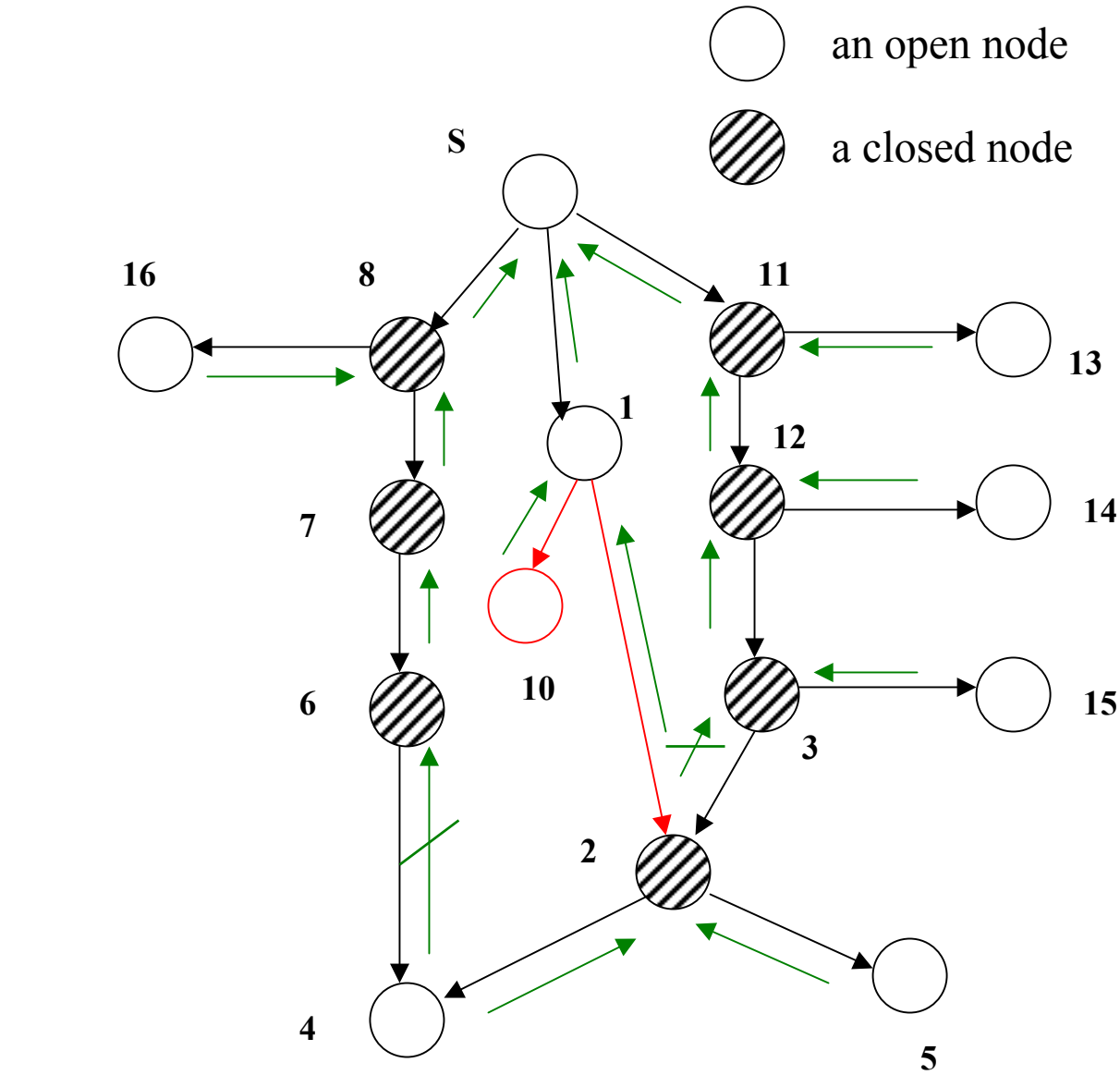
7 Establish a pointer to n from those members of M that were not already in G (i.e. not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member of M that was already on OPEN or CLOSED, decide whether or not to redirect its pointer to n. (See text.) For each member of M already on CLOSED, decide for each of its descendants in G whether or not to redirect its pointer. (Note: You have to check all descendants, not just the immediate children.)

8 Reorder the list OPEN, either according to some arbitrary scheme or according to heuristic merit.

9 Go LOOP.



A search tree using an evaluation function.



Open List

Closed list

f(n)

4	1	S
5	4	2
.	5	3
.	.	6
.	10	.
		.
		1

f(n) estimate of how much work is involved in a solution which has node n on its path.

Admissibility:

If algorithm is admissible it always finds an optimal path from S to goal (solution).

An algorithm is admissible if

$$h(n) \leq h^*(n)$$

actual cost for the solution path from n to goal.

Informed algorithms Vs. Less informed algorithms:

Two algorithms with $h_1(a)$ and $h_2(n)$ respectively (both heuristic search): if $h_1(n) \leq h_2(n) \leq h^*(n)$

then algorithm 2 using is more informed than algorithm 1.

Algorithm 2 will always do less work than algorithm 1 in finding the solution to the problem.

Monotonicity:

If we have the following condition for a heuristic search algorithm:

(for any two nodes n_i and n_j):

$$h(n_i) - h_2(n_j) \leq C(n_i, n_j)$$

Cost from n_i to n_j

goal node $h(t) = 0$

Thus when A* expands a node it has already found the best path to that node, therefore there is no need to update backpointers.

Minimax control:

Search into the search space a few levels forward, and make the best move possible (this may not be necessarily the winning move).

Assume we look forward 2 levels into the search space. Do the following:

(1) Conduct a breadth first search until all nodes of level 2 are generated.

(2) Apply an evaluation function to all the leaf nodes.

(3) Propagate the values backward to make selection of next move.

<<< Refer to Section 6.2, pg. 165 of the text >>>

Ex: take the tic-tac-toe problem.

	X	0
0	X	
	X	

Name the first player max, the other player min.

f(n) = (no of complete rows, columns, or diag. still available to max) - (no of complete rows, columns, or diag. still available to other player, min).

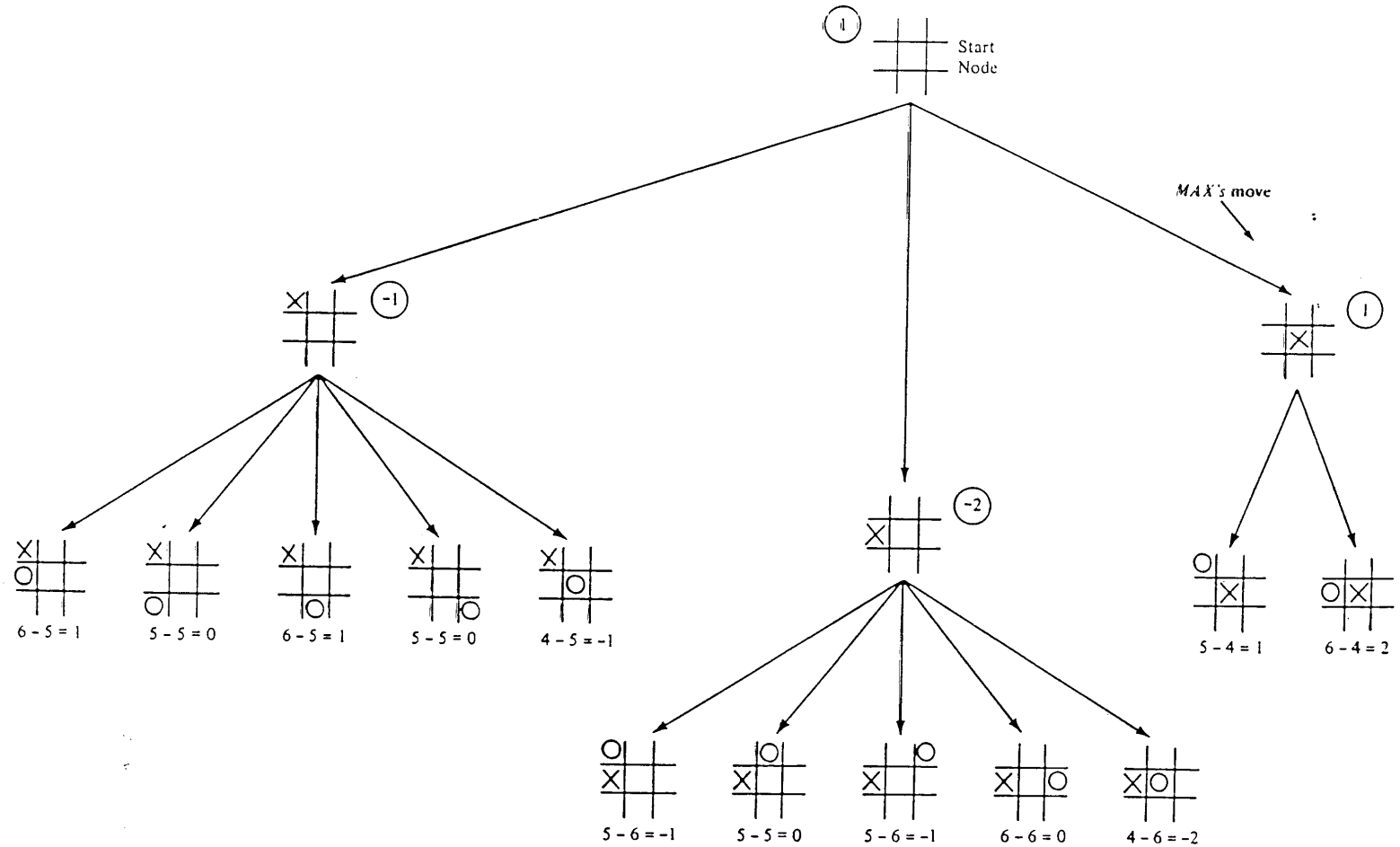
f(n)= ∞ (if no winning position)

f(n) =?

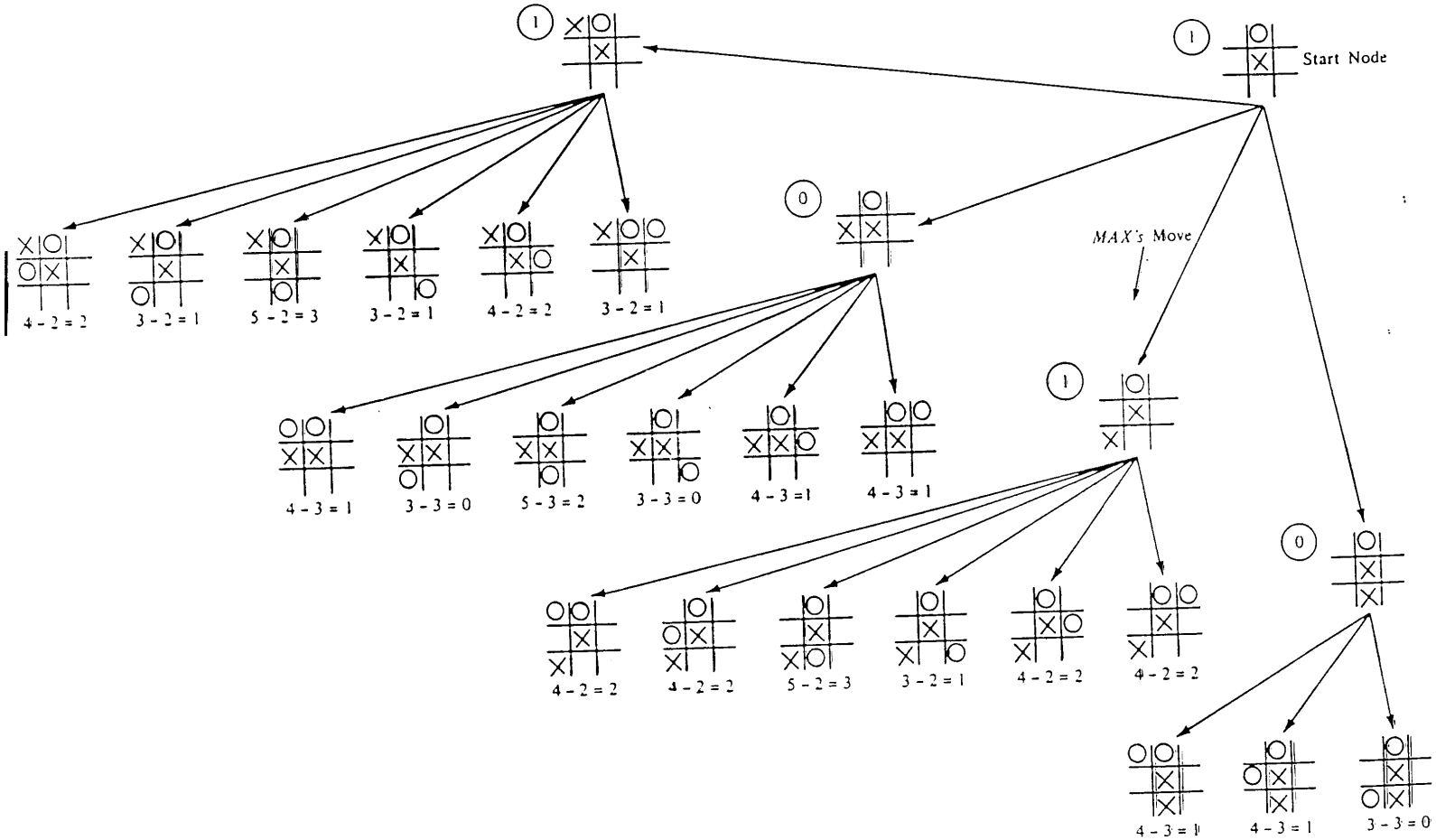
= 5- 4 =1

n =

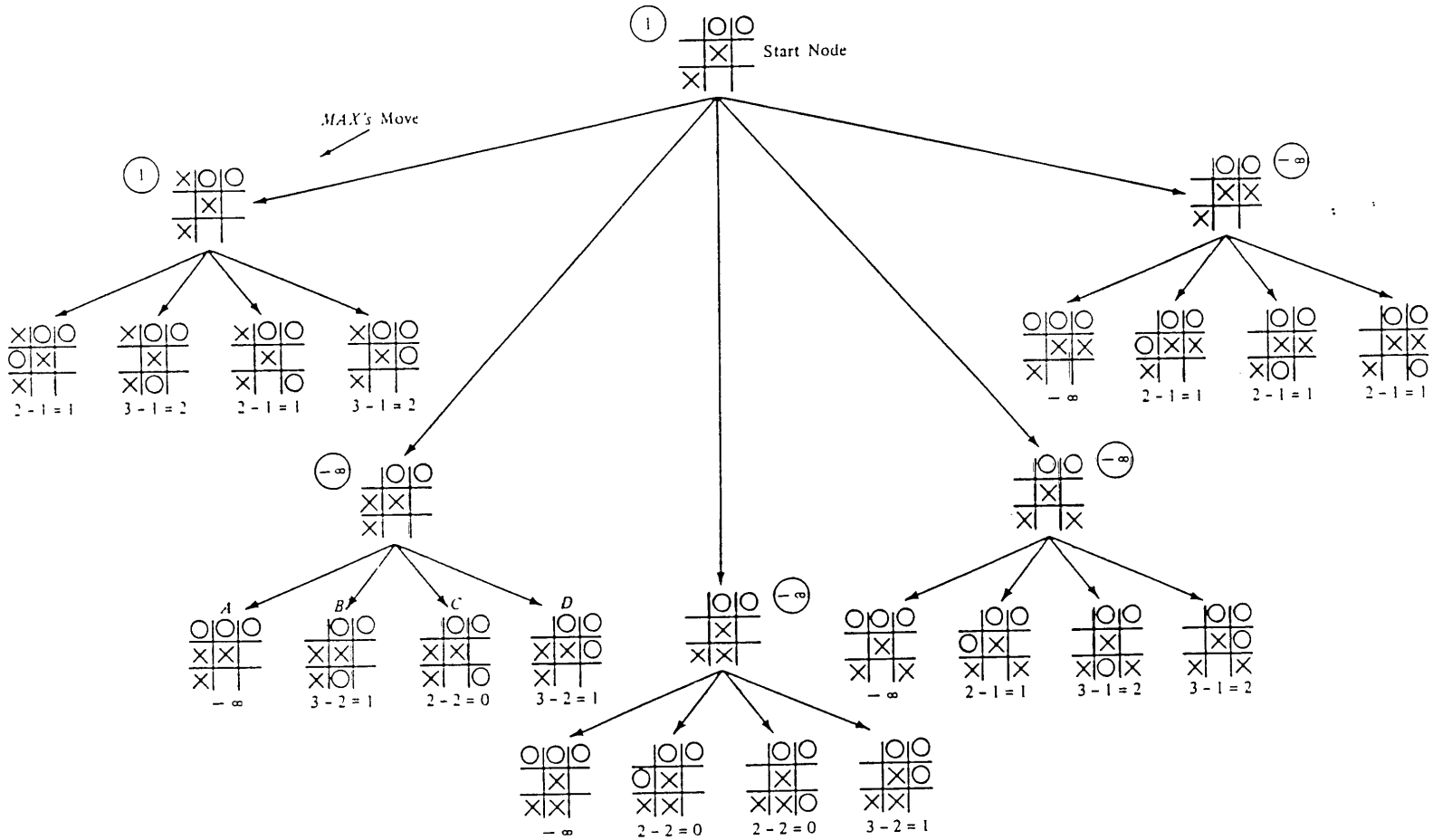
0		
	X	



Minimax applied to tic-tac-toe (stage 1).



***Minimax applied to** tic-tac-toe (stage 2).*



Minimax applied to tic-tac-toe (stage 3).

α-β Procedure

<<< Refer to Section 6.3 of the text >>>

Instead of enumerating all nodes K levels ahead, we only generate the nodes that could affect the outcome. Therefore obtain computational efficiency.

The values assigned to MAX as nodes are generated are denoted α values. Similarly, values assigned to Min as nodes are generated, are denoted as β values.

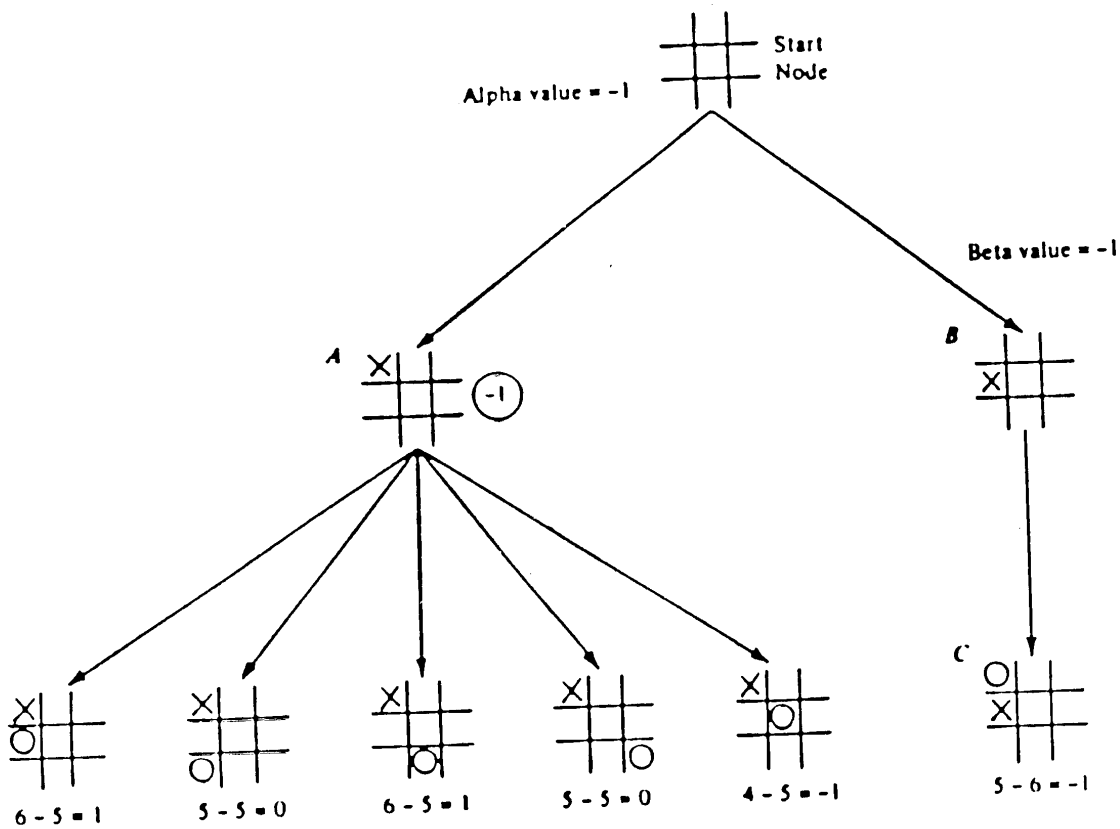
Alpha-Beta Procedure:

(a) Set the alpha value of a MAX node equal to the largest final backed up value of its successors.

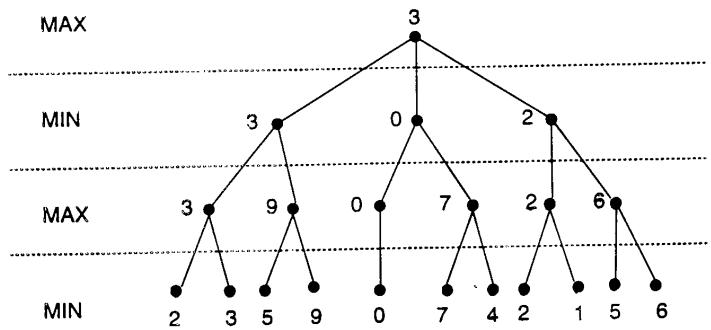
(b) The beta value for a Min node is at equal to the smallest backed up value of its successors.

(c) Discontinue search at any Min node having beta value less than or equal to the alpha value of its MAX ancestors.

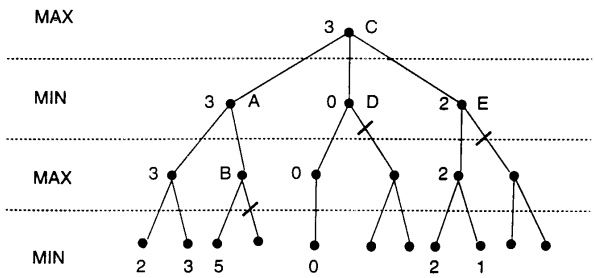
(d) Discontinue search below any MAX node having alpha value greater than or equal to Beta value of any Min node ancestors.



f = e(X) - e(0)
e(X) = no. of complete rows, columns, or diagonals from which X can still win.
e(0) = same for 0.



Minimax to a hypothetical state space. Leaf states show heuristic values; internal states show backed-up values.



A has $\beta = 3$ (A will be no larger than 3)
B is β pruned, since 5 > 3
C has $\alpha = 3$ (C will be no smaller than 3)
D is α pruned, since 0 < 3
E is α pruned, since 2 < 3
C is 3

Alpha-beta prune applied to the state space of previous figure. States without numbers are not evaluated.