Artificial Intelligence
Representation and Search Techniques

L. Manevitz

Goals of Lecture

- Representing Problems :
  - Various Issues and Considerations.
  - Production Systems.
- Production systems :
  - State Space.
  - Goals.
  - Transformation Rules.
  - Control (Search Techniques).

Elements of Production Systems

- Representation :
  - State Space.
  - Goal States.
  - Initial States.
  - Transformation Rules.
- Search Algorithms :
  - Uninformed.
  - “Heuristic”.

Examples of Problems

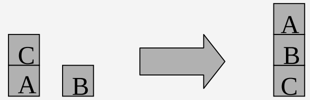
- “Toy” Problems :
  - Water jug.
  - Cannibals.
  - 8 – Queens.
  - 8 Puzzle.

Examples of Problems cont.

- <u>"Real" Problems :</u>
  - Schedules.
  - Traveling Salesman.
  - Robot navigation.
  - Language Analysis (Parsers, Grammars).
  - VLSI design.

Points to Consider when Representing Problems

- Decomposable ?



- Can partial steps be ignored or undone ?

Theorem proving vs. chess

- Predictable ?

Bridge

Points to Consider when Representing Problems cont.

- Is "good" solution easily recognizable ?

Traveling Salesman

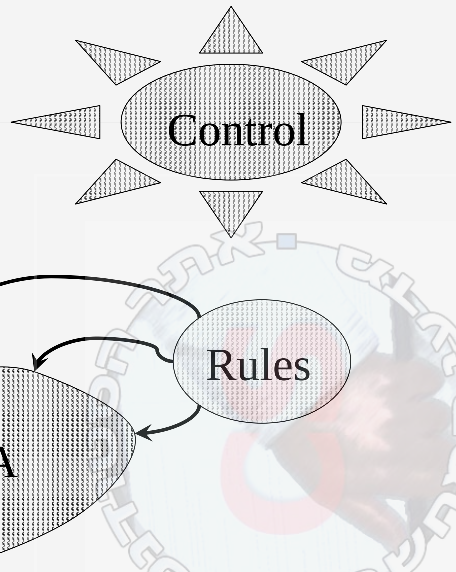
- Is Knowledge Base consistent ?

Big Data Base and limitations of logic

- How much Knowledge is needed ?

Chess, Newspaper-Understanding

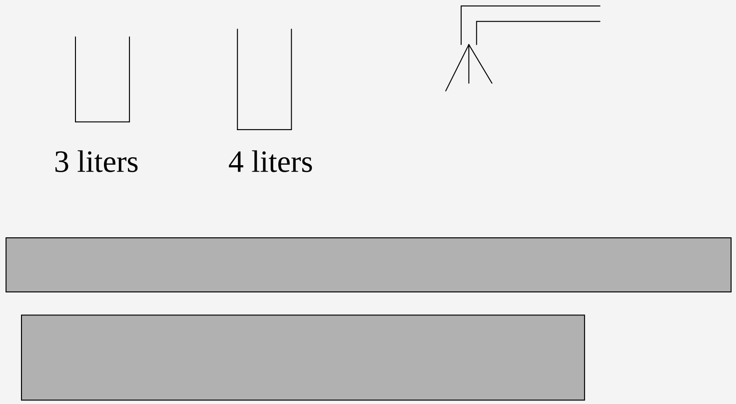
- Stand–alone vs. Inter–active.



Issues In Representing Problem

1) Choice of representation of Data Base.
    1) Specify initial states
    2) Specify goal states
2) Appropriate Rules.
    1) Issues:
        1) Assumptions in problem
        2) How general
        3) How much work pre-computed and put into rules
3) Control (later)

Water Jug Problem



Water Jugs cont.

- Rules :
    - <x,y> x<4 → <4,y> (Fill 4 liters)
    - <x,y> y<3 → <x,3> (Fill 3 liters)
    - <x,y> → <o,y> (Dump 4 liters)
    - <x,y> → <x,0> (Dump 3 liters)
    - <x,y> | x+y >= 4 → <4,y-(4-x)>
    - <x,y> | x+y >= 3 → <x-(3-y),y>
    - <x,y> | x+y <= 4 → <x+y,0>
    - <x,y> | x+y <= 3 → <0,x+y>

Example no.1

- **CHESS :**
    - R1 – If pawn not blocked then move it one space forward.
    - R2 – If pawn in original position and not blocked for two spaces move it two spaces.
    - Etc.

Example no.2

- **SPEECH**:
  - R1 – If input not analyzed try and identify phonemes.
  - R2 – Take some possible syllables and try and form words.
  - Etc.

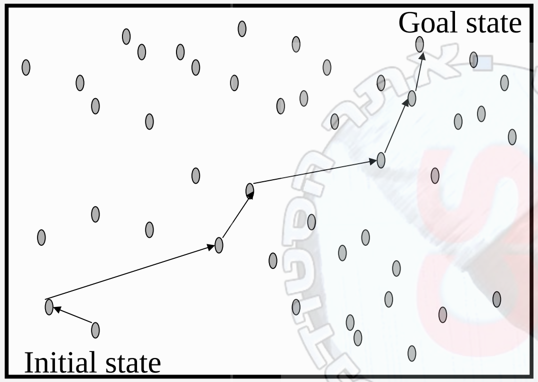
Production

1. DATA ← initial database.
2. Until DATA satisfies termination condition do:
3. begin:
   - ➢ Select some rule, R, in the set of rules that can be applied to DATA.
   - ➢ DATA ← result of applying R to DATA.
4. end.

Problem Description

1) Define State Space containing all possible configurations of relevant objects.
2) Specify some states as initial states.
3) Specify some states as goal states.
4) Specify Rules

Search Through State-Space



Control Strategies

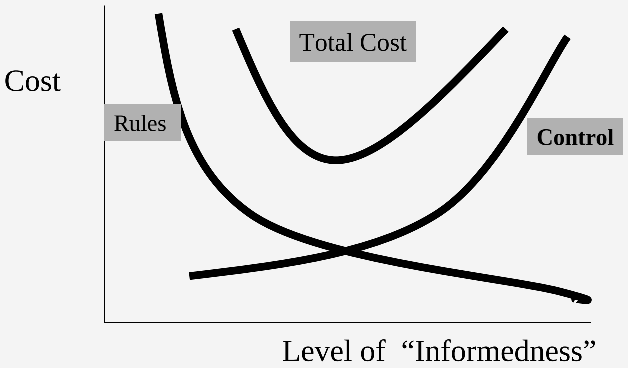
- What do we want?
  - Cause motion
  - Be systematic

  Examples
  - Breadth first
  - Depth first
  - Back Tracking
  - Hill Climbing
  - Best First

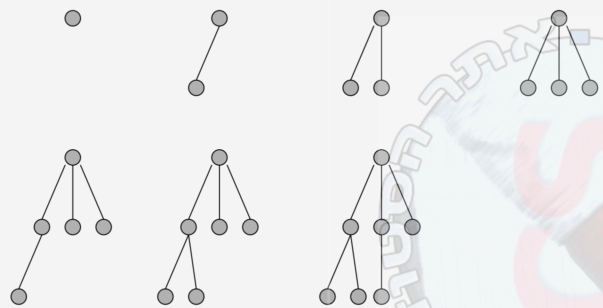
Control via Search Techniques

- <u>“Uninformed” :</u>
  - Breadth – First.
  - Depth – First.
  - Backtracking.
- <u>“Informed” – Heuristic :</u>
  - Hill Climbing.
  - Best First, A*.

Costs of Production System



Breadth First cont.



Breadth First

b = branching factor d=depth

$1 + b + b^2 + \ldots + b^d$

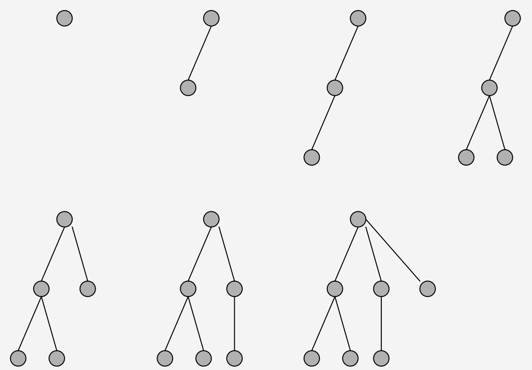
Time Complexity – $O(b^d)$

Space Complexity – $O(b^d)$

Breadth First fix exponents

Depth	Nodes	Time	Memory
2	111	.1 sec	1 kb
4	11,111	11 sec	1 mega
6	10**6	18 min	111 mega
8	10**8	31 hours	11 giga
10	10**10	128 days	1 tera
12	10**12	35 years	111 tera
14	10**14	3500 years	11,111 tera

Depth First cont.

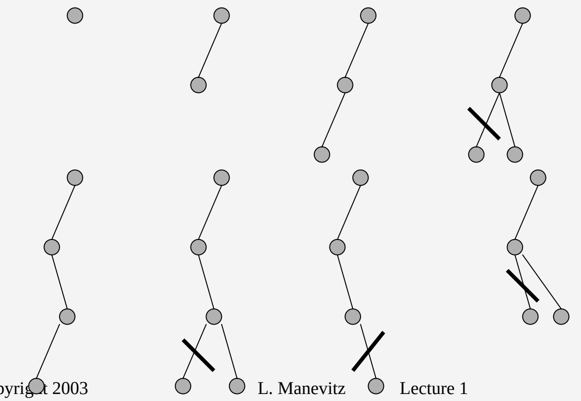


Depth First (some adjustments needed for depth bound)

DEPTH FIRST (INITIAL DATA)

1. DATA ← INITIAL DATA
2. IF DATA = GOAL THEN EXIT WITH NULL
3. RULES ← APPRULES (DATA)
4. IF RULES = NULL EXIT WITH FAILURE
5. R ← FIRST (RULES)
6. DATA ← R (DATA)
7. IF DATA AT GOAL EXIT WITH R
8. PATH ← DEPTH FIRST (DATA)
9. IF PATH = FAILURE EXIT WITH FAILURE
10. EXIT WITH R^PATH

Backtracking cont.



Backtracking Algorithm

BACKTRACK (DATA)

1. IF TERMINAL (DATA) RETURN NULL
2. IF DEADEND (DATA) RETURN FAIL
3. RULES ← APPRULES (DATA)
4. LOOP : IF RULES = NULL RETURN FAIL
5. R ← BEST (RULES,DATA)
6. RULES ← RULES – {R}
7. NEWDATA ← R (DATA)
8. PATH ← BACKTRACK (NEWDATA)
9. IF PATH = FAIL THEN GO TO LOOP
10. RETURN R^PATH

Backtracking

- Works like Depth First.
- Stores only last path.
- <u>Disadvantages :</u>
  - May never terminate –
    - New nonterminal database always generates.
    - Cycle.
  - However can apply heuristics to choose –
    - Best rule.
    - Better heuristics – less backtracking.

Backtracking cont.

Try a rule – if not successful go back and try a different one.

Example – try rules in this order: L, U, R, D.

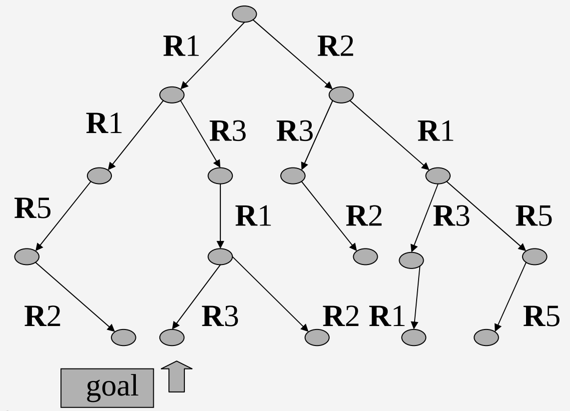
<u>Back up if –</u>

1. Repeat position on path – back to initial / state.
2. Whenever have already applied # of rules – depth bound.
3. When no more rules can be found.

Backtracking1 Algorithm (checking for loops)

```
BACKTRACK1 (DATALIST)
DATA ← FIRST (DATALIST)
IF MEMBER (DATA,TAIL(DATALIST)) RETURN FAIL
IF TERMINAL (DATA) RETURN NULL
IF DEADEND (DATA) RETURN FAIL
IF LENGTH (DATALIST) > BOUND RETURN FAIL
RULES ← APPRULES (DATA)
LOOP : IF RULES = NULL RETURN FAIL
R ← FIRST (RULES)
RULES ←RULES -R
RDATA ← R (DATA)
RDATALIST ← CONS (RDATA,DATALIST)
PATH ← BACKTRACK1 (RDATALIST)
IF PATH = FAIL THEN GO TO LOOP
RETURN CONS (R,PATH)
```


Search Tree



8 Puzzle

- Problem :

8	7	5
4	1	3
6		2

→

1	2	3
8		4
7	6	5

- Data Base : 3x3 Matrix or a Vector with length 9.
- Rules : Precondition :

Rules	Precondition
1) Move Blank Left	Not at extreme left
2) Move Blank Right	Not at extreme right
3) Move Blank Up	Not at top row
4) Move Blank Down	Not at bottom row

8 Puzzle cont.

- <u>Initial State :</u>

2	8	3
1	6	4
7		5

Move Blank Up
Move Blank Up
Move Blank Left
Move Blank Down
Move Blank Right

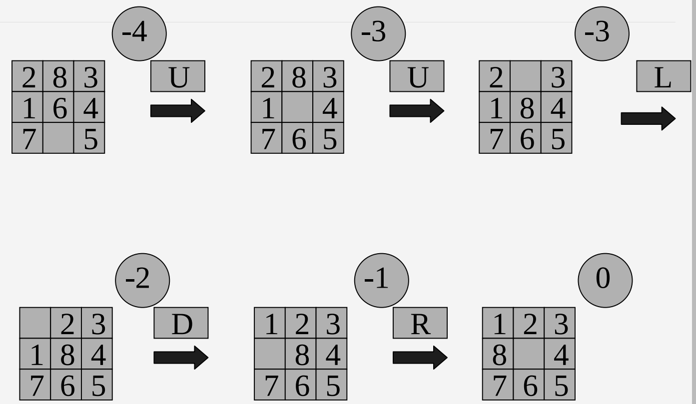
- <u>Goal State :</u>

1	2	3
8		4
7	6	5

Hill Climbing

1) Use heuristic function as measure of how far off the number of tiles out of place.
2) Choose rule giving best increase in function.

Example



Hill Climbing cont.

Problems :

1) Local Maxima –

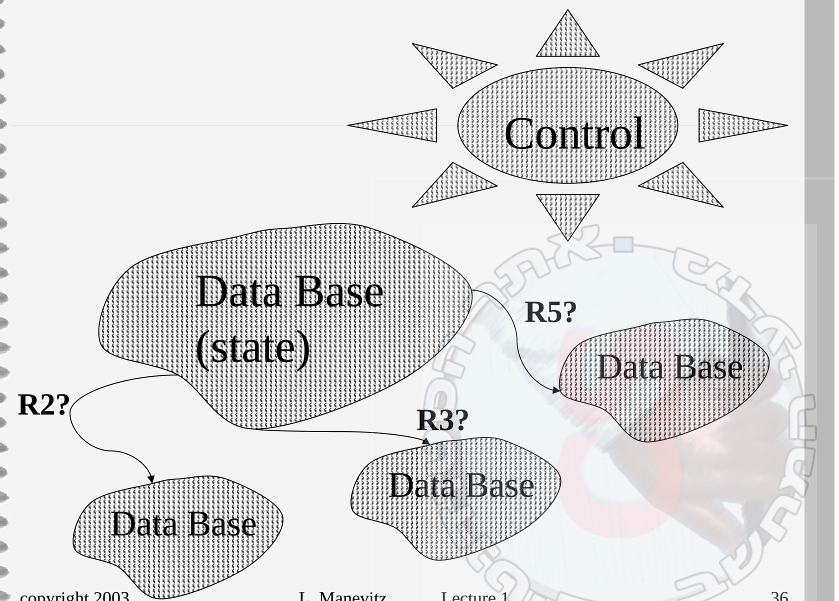
Initial

1	2	5
	7	4
8	6	3

Goal

1	2	3
	7	4
8	6	5

2) Plateaus.
3) Ridge.



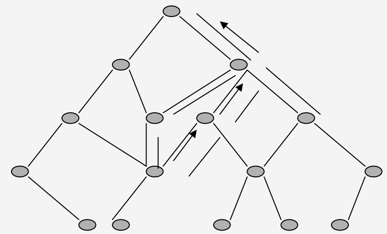
Graphs - Digraphs

- Graph – Nodes and Edges.
- Digraph – Nodes and directed arcs.
- Tree – each Node has one parent :
  - Root – no parent.
  - Leaf – no successors.
- Path – $n_1 \ldots n_k$.

Graphs – Digraphs cont.

Implicit vs. Explicit Graph

generally, make explicit sub-graph of implicit graph.



Implicit Graph

Explicit Graph

Graph Search – Data Structures

- 2 List of “discovered nodes”
  - Open (not yet “expanded”)
  - Closed (already “expanded”)
- Graph
- Pointers on Graph (like “Hansel and Gretel”)

- The graph grows as nodes are expanded
  - Explicit versus Implicit Graph

Graph Search

1. G ← s ; Open ← s ; Closed ← NULL
2. <u>LOOP</u>: EXIT WITH FAILURE IF Open=NULL.
   1. Take first node n from Open , Add to Closed.
   2. If n is goal exit with SUCCESS.
      (solution obtained by pointer path from n to s).
   3. Expand n : generate M set of Successors + add to G as successors to n.

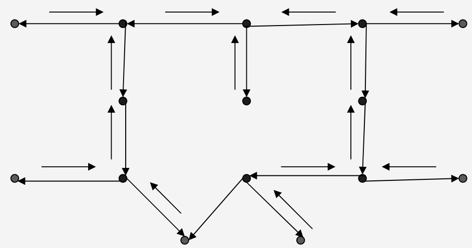
Graph Search

4. For each m from M :
    1. If m is new add to Open and pointer back to n.
    2. If m is already on Open, see if to redirect pointer to n.
    3. If m already on Closed, see if to redirect pointer to n. Then check all descendants as well.
5. Reorder Open.
6. Go to LOOP.

Example

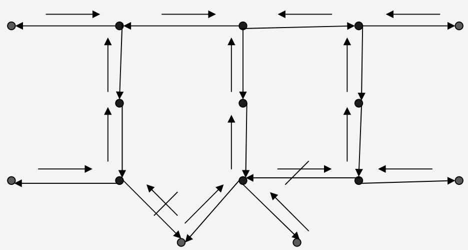
A node with not yet checked successors

A node with checked successors



Example cont.

The changes in the graph are :



Disadvantage of Graphs

1) Have to verify that a new node hasn’t appeared before (expensive).

2) If don’t verify – then redundant successor computations.

Ordering of Nodes (in Graph Search)

- Descending Order (expand deepest first) –
  Depth First Search (with cut-off).

- Ascending Order –
  Breadth First Search.

- Heuristic Best First -
  $h(n) = g^*(n) + d^*(n)$
  A* Algorithm

Tree Search

- Form a Queue consisting of root node
- Until Queue = Null or Goal achieved
  - See if 1st element is goal. Do nothing if yes
  - If not, remove element from queue and add its children
    - in back of queue (breadth-first)
    - in front of queue (depth-first)
    - re-sort queue (best-first)
    - front of queue (sorted by estimate) hill-climbing

A* Algorithm

- Two Lists –
  - Open.
  - Closed.
- Parent List.
- Heuristic Function

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

(cost of solution constrained through n)

A* Algorithm cont.

- Open = {s}; $g^*(s) = 0$; $f^*(s) = h^*(s)$
  Closed = NULL
- Open = NULL → Return Failure
- Bestnode ← Best (Open)
- Open ← Open – {Bestnode}
- Goal (Bestnode) → Return Solution
- Successors ← Successor (Bestnode)

A* Algorithm cont.

- For each S $\in$ Successors Do:
  - Parent (S) ← Bestnode
  - g*(S) = g*(Bestnode) + c(Bestnode,S)
  - Does S $\in$ Open ? (i.e. identify with OLD)
    - Add OLD to Children (Bestnode)
    - If g*(S) < g*(OLD)
      - g*(OLD) ← g*(S)
      - Parent (OLD) ← Bestnode
      - f*(OLD) ← g*(OLD) + h*(OLD)

A* Algorithm cont.

- Does S $\in$ Closed ?
  - No

    Add S to Children (Bestnode)
  - Yes (identified with OLD)

    Add OLD to Children (Bestnode)
  - If g*(S) < g*(OLD)
    - g*(OLD) ← g*(S)
    - Parent (OLD) ← Bestnode
    - f*(OLD) ← g*(OLD) + h*(OLD)

A* Algorithm cont.

- Propagate change downwards
- Do Depth First traversal from OLD
- Terminate if g*(node) <= path from OLD
  or w/o successors
  or on Open

- Otherwise change g*(node)
  f*(node)
  change Parent (node)

Optimality of A*

- Heuristics: f(x) = g(x) +h(x)
- Here g(x) is estimate of g*(x) the actual cost to get to x from s.
- h(x) is estimate of h*(x) the miniml cost to get to any goal from x.
  - Choose g(x) to be cost in EXPLICIT GRAPH
  - Choose h(x) such that h(x) <= h*(x)

- In such a circumstance A* returns optimal path.

- Examples:
- $h(x) = 0$
- For 8 puzzle $h(x) =$ number of tiles in wrong position
- For 8 puzzle $h(x) =$ sum of distances each tile is from home
- sequence scores $P(x) + 3\ S(x)$

Proof of Optimality

ON Blackboard.

Main points: If doesn't terminate,

eventually all points in open have very large f value since they will have large g value.

But always a point in OPEN on optimal path;

thus f value bounded.

Moreover just prior to termination, must choose a node with small f value; so cant terminate erroneously.

AND/OR GRAPHS

- Hypergraphs.
- Hyperarcs connects node with SET of nodes.
- Connectors are 1-ary, 2-ary and so on.

Solution subgraph G' of G from node n to N terminals

- If n in N, G' is just n
- If n has outgoing connector to set of nodes

a, b, c … such that there is solution graph from each of a, b ,c separately to N;

then G' consists of n, connector, a, b, c, …

and each of the solution graphs from a, b,c…

Otherwise no solution graph exists

Costs

- Include cost of connector.
- Then cost from node n to N, k(n,N) is defined recursively by
  - if n in N k(n,N) = 0
  - n has connector to a, b, c … in solution graph with cost c
    - k(n,N) = c + k(a, N) + k(b, N) + k (c, N) + …

- Create a search graph G, consisting solely of start node, s. Put s on list OPEN
- Create a list called CLOSED = Null
- LOOP: if OPEN = Null, exit FAILURE
  - Move 1st node of OPEN, n, to CLOSED
  - If n GOAL exit with solution via pointers from n to s. (pointers placed later)
  - Let M be set of successors of n, place in G
  - Make a pointer to n from members of M not already in G. Add these members to OPEN.
  - For members of M already on OPEN decide if to change pointer to n.
  - For members of M already on CLOSED
    - Decide if to change pointer to n
    - Decide for each of its descendents in G whether to change pointer

Reorder OPEN

GO LOOP