Reid G. Smith. *Artificial Intelligence – CS375A – Notes*.
Dalhousie University, Halifax, NS, Canada, Fall 1981.

Can Machines Think?

… how could you tell?

The Turing Test

What is a computer?

Is computation intrinsically tied up with numbers?

A Symbol Processor

What is Intelligence?

1) Formulate Goals and Subgoals
2) Use Symbols and Abstractions
3) Use Vast Amounts of Knowledge about the Environment
4) Learn from the Environment
5) Exhibit Self-Awareness and a Sense of Self

Is Intelligence Ephemeral?

You regard an action as intelligent until you understand it. In explaining, you explain away

What is Artificial Intelligence?

1) The attempt to construct artifacts that exhibit behavior we call “intelligent behavior” when we observe it in human beings, *or*, The development of a systematic theory of intellectual processes, wherever they are found.

2) Theoretical psychology

Application Areas

Core Topics

- Heuristic Search
- Knowledge Representation
- Common-Sense Reasoning & Problem Solving
- Systems & Languages

Game Playing	Expert Systems
Theorem Proving	Automatic Programming
Robotics	Machine Vision
Natural Language Systems	Information Processing Psychology

Can Machines Think – Objections:

The Theological Objection

Thinking is a function of man's immortal soul. God has given an immortal soul to every man and woman, but not to any other animal or to machines. Hence no animal or machine can think.

The "Heads in the Sand" Objection

The consequences of machines thinking would be too dreadful. Let us hope and believe that they cannot do so.

The Mathematical Objection

Based on Gödel's theorem (1931) – In any sufficiently powerful logical system, statements can be formulated which can neither be proved nor disproved within the system. Therefore there are limitations on the powers of any particular machine, but no such limitations apply to the human intellect.

Can Machines Think – Objections:

The Argument from Consciousness

Not until a machine can write a sonnet of compose a concerto because of thoughts and emotions felt, and not by the chance fall of symbols, could we agree that machine equals brain – that is, not only write it but know that it had written it. No mechanism could feel (and not merely artificially signal, an easy contrivance) pleasure at its successes, grief when its valves fuse, be warmed by flattery, be made miserable by its mistakes, be charmed by sex, be angry or depressed when it cannot get what it wants.

Arguments from Various Disabilities

I grant that you can make machines do all the things you have mentioned but you will never be able to make a machine do X. (X ← be resourceful, make mistakes, fall in love …)

Can Machines Think – Objections:

Lady (Ada) Lovelace's Objection (1842)

The Analytical Engine has no pretensions to *originate* anything. It can do *whatever we know how to order it to perform*.

The Argument from Informality of Behavior

It is not possible to produce a set of rules purporting to describe what a man should do in every set of circumstances. Therefore men cannot be machines.

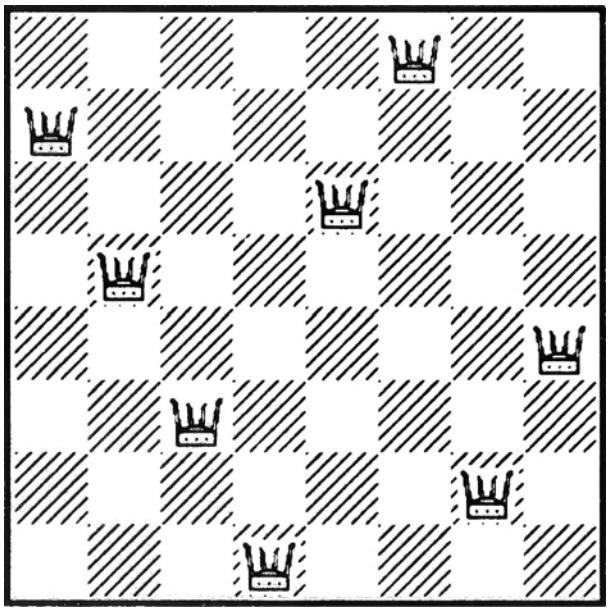
The Argument of Super Excellence

Computer composition compares miserably with that of Mozart or Chopin. Therefore …

Heuristic Search

1) Transform a *problem* into the canonical problem of finding a path through a *space* of problem states from the initial state to a goal (i.e., solution) state.

2) *Reduce* a problem to various subproblems that are also *reduced* in turn (and so on) until the ultimately resulting subproblems have trivial or known solutions.



Problem: How can the effects of the *combinatorial explosion* of exhaustive search be lessened?

… Heuristics.

Knowledge Representation

How should knowledge be acquired and represented so that it can best be used by a computer system?

Types of Knowledge

objects (facts about the environment)

Terminals have keyboards

events

The system crashed last night

performance of skills

I know how to type

meta-knowledge

I know that I understand LISP code

Survey of Representational Techniques

State Vector

(2 4 6 8 3 x x x)

Logic

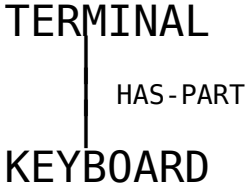
All terminals have keyboards
$\forall$ x. Terminal (x) $\rightarrow$ HasKeyboard (x)

Procedural Representation

What types of terminal have 24 line screens?

```
(For x in TERMINAL-TYPES collect x
   when (LINES x) = 24)
```

Semantic Networks



Production Systems

IF *testing for equality* **AND** *items are atoms*
THEN *use EQ*

FRAMES and UNITS

Previous experience and expectations. Data structures for *representing* stereotyped objects, situations, and events, and for *organizing* knowledge.

Slot: How what you know fits into the context created by the unit. An encoding of structural information.

Example:

Unit: ELEPHANT

Generalization: MAMMAL

```
COLOR:            Gray
TEXTURE:          Wrinkled
WEIGHT:           An integer (DEFAULT 2000 Lbs)
HEIGHT:           1-6 M
STYLE-OF-EARS:    Sleek, Floppy, ...
```

Unit: CLYDE

Generalization: ELEPHANT

```
COLOR:            Gray
TEXTURE:          Wrinkled
WEIGHT:           2000 Lbs
HEIGHT:           3 M
STYLE-OF-EARS:    Sleek
```

Aspects or Fields: Different restrictions on what can fill a slot (e.g., datatype, allowable range of values). Advice on how to fill a slot: **Procedural Attachment** (*To-Fill, If-Changed,* …)

Encoding Declarative Knowledge and Procedural Knowledge

Generalization Hierarchy: A further encoding of structure.

FRAMES and UNITS

```
ANIMAL
    MAMMAL
        ELEPHANT
        SPERM-WHALE
    SHARK
```

Property Inheritance: An approach to the **Symbol-Mapping Problem** – Given it is learned that FLOSSIE is an elephant, a number of facts become "accessible." How to make that happen? Antecedent theorems are troublesome. Which properties to assert? Consequent theorems are tricky. Which subgoals to pursue?

How to *map* the canonical elephant into FLOSSIE?

Structured Object Representations

```
Generic TERMINAL Frame
  Lines: An Integer
  Characters/Line: An Integer
  Type: One of [VT100 DM2500]

MY-TERMINAL Frame
  Lines: 24
  Characters/Line: 80
  Type: DM2500
```

Common-Sense Reasoning and Problem Solving

1) How should programs deduce facts that are implied by other explicitly represented facts but are not themselves explicitly represented?

2) How should plans of actions to achieve given goals be generated and executed, replanning as necessary?

Puzzle Solving

Logic Theorist

Given:

$(p \wedge p) \rightarrow p$

$p \rightarrow (q \wedge p)$

$(p \wedge q) \rightarrow (q \wedge p)$

$[p \wedge (q \wedge r)] \rightarrow [q \wedge (p \wedge r)]$

$(p \rightarrow q) \rightarrow [(r \wedge p) \rightarrow (r \wedge q)]$

Prove:

$\neg(p \wedge q) \rightarrow \neg p$

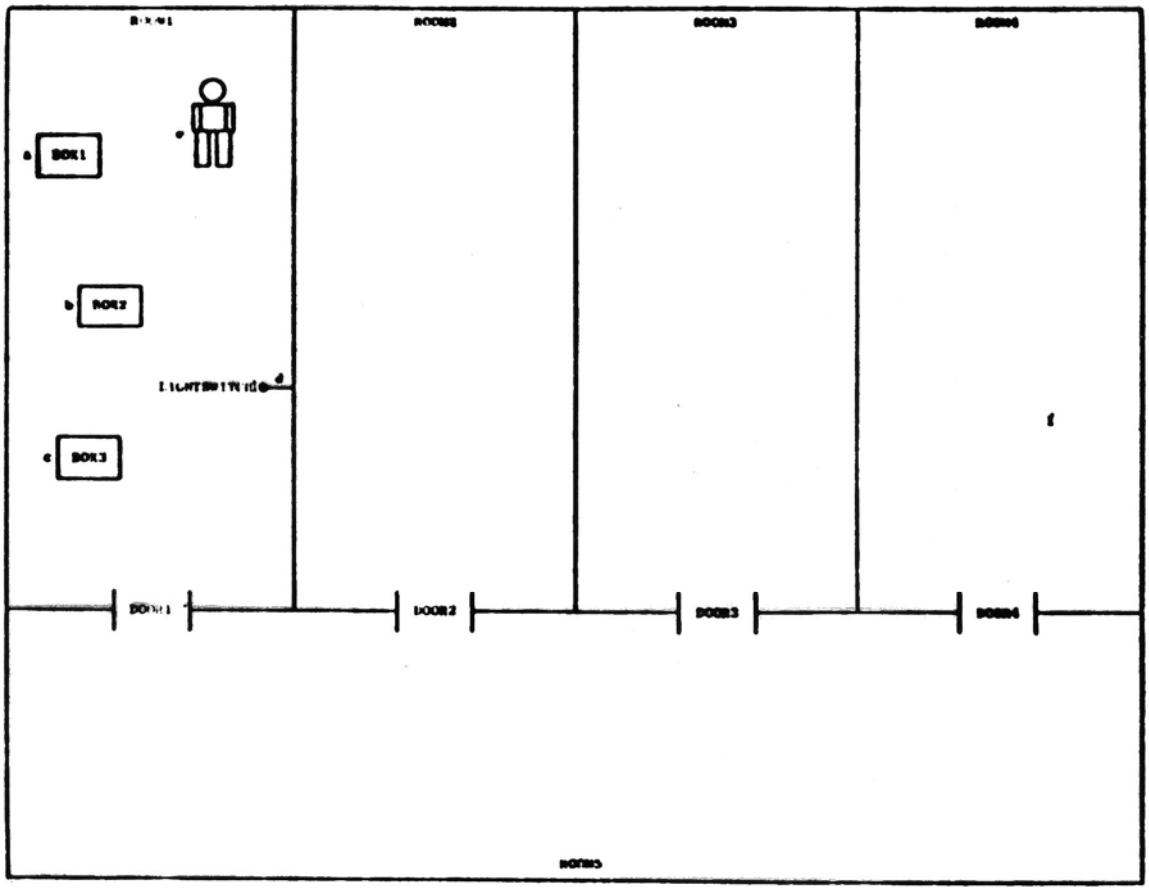
Reasoning Backward, Means-Ends Analysis

Early Robotics

STRIPS

Robot called Shakey
Solving Problems in a Multi-Room Environment

Planning
Execution Monitoring / Replanning
Hierarchical Planning



Systems and Languages

How should important strategies, processing methods, and representations be incorporated into more powerful and useful programming languages?

Lisp and IPL-V
list processing

QA4, QLISP, PLANNER, CONNIVER
search, pattern-matching, contexts
pseudo-parallel control regimes
(e.g., generators)

INTERLISP Programming Environment
DWIM, File Package, Programmer's Assistant
Spaghetti Stack, Break Package

Lisp Machines
tailored microcode, efficient list storage
~ $70k

Tools
Timesharing
Screen-based editors: E, EMACS, TV-EDIT
Document Compilers: PUB, SCRIBE
Bit-Mapped Graphics

Application Areas

Game Playing

Checkers (Samuel)
Chess (CHESS 4.6 – rating: 2271 [master rating])
Backgammon (BKG)
… Man/Machine differences

Expert Systems

MYCIN / DENDRAL / PUFF / MOLGEN
Transfer of Expertise: Human → Machine

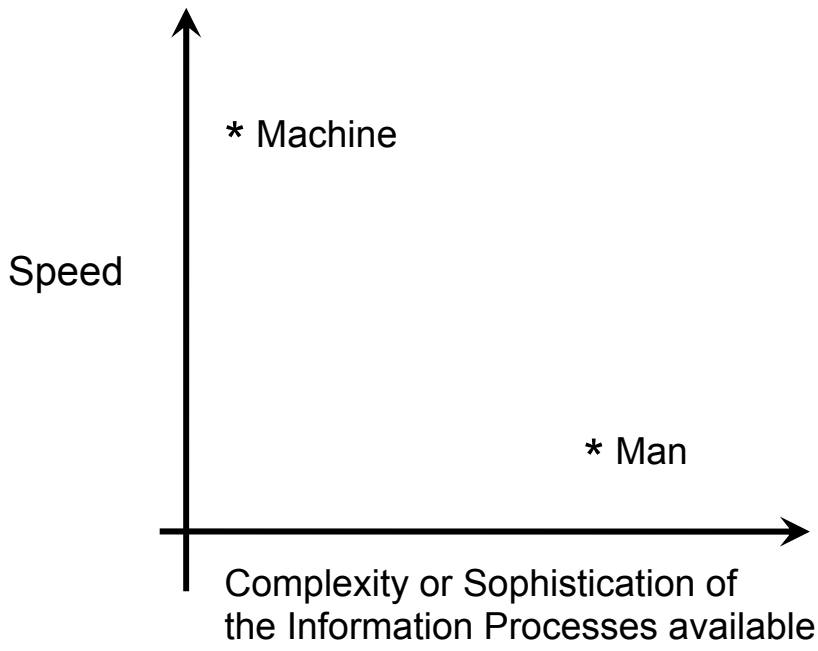
MACSYMA

$$y^{2x} - 3y^{x} + 2 = 0 \quad (\text{for } x)$$

$$\int (x + e^{x})^{2} dx$$

Theorem Proving

Resolution – single rule of inference
Man/Machine Systems


* Machine
Speed
* Man
Complexity or Sophistication of
the Information Processes available

```
--------PATIENT-1--------
  1) Patient's name: (first last)
  **  FRED SMITH
  2) Sex:
  **  MALE
  3) Age:
  **  55
  4) Have you been able to obtain positive
     cultures from a site at which Fred Smith
     has an infection?
  **  YES
  --------INFECTION-1--------
  5) What is the infection?
  **  PRIMARY-BACTEREMIA
  6) Please give the date and approximate time
     when signs or symptoms of the primary-
     bacteremia (INFECTION-1) first appeared.
     (mo/da/yr)
  **  MAY 5, 1975
     The most recent positive culture associated
     with the primary-bacteremia (INFECTION-1)
     will be referred to as:
  --------CULTURE-1--------
  7) From what site was the specimen for
     CULTURE-1 taken?
  **  BLOOD
  8) Please give the date and time when this
     blood culture (CULTURE-1) was obtained.
     (mo/da/yr)
  **  MAY 9, 1975
     The first significant organism from this
     blood culture (CULTURE-1) will be referred
     to as:
  --------ORGANISM-1--------
  9) Enter the identity of ORGANISM-1:
  **  UNKNOWN
  10) Is ORGANISM-1 a rod or coccus (etc.)?
```

** ROD
11) The gram stain of ORGANISM-1:
** GRAMNEG
12) Have there been POSITIVE cultures from sites which are NOT associated with infections about which you are seeking advice?
** NO
13) Did ORGANISM-1 grow aerobically?
** YES (.8)

...

My therapy recommendation will be based on the following probable infection(s) and potential causative organism(s):

INFECTION-1 is PRIMARY-BACTEREMIA

<Item 1> The identity of ORGANISM-1 may be PSEUDOMONAS-AERUGINOSA
<Item 2> The identity of ORGANISM-1 may be KLEBSIELLA-PNEUMONIAE
<Item 3> The identity of ORGANISM-1 may be E.COLI
<Item 4> The identity of ORGANISM-1 may be BACTEROIDES-FRAGILIS
<Item 5> The identity of ORGANISM-1 may be ENTEROBACTER
<Item 6> The identity of ORGANISM-1 may be PROTEUS-NON-MIRABILIS

34) Was penicillin added to this blood culture (CULTURE-1)?

** **WHY**

[i.e., WHY is it important to determine whether penicillinase was added to CULTURE-1?]

[3.0] This will aid in determining whether ORGANISM-1 is a contaminant.

It has already been determined that

[3.1] the site of CULTURE-1 is blood, and

[3.2] the gram stain of ORGANISM-1 is grampos

Therefore, if

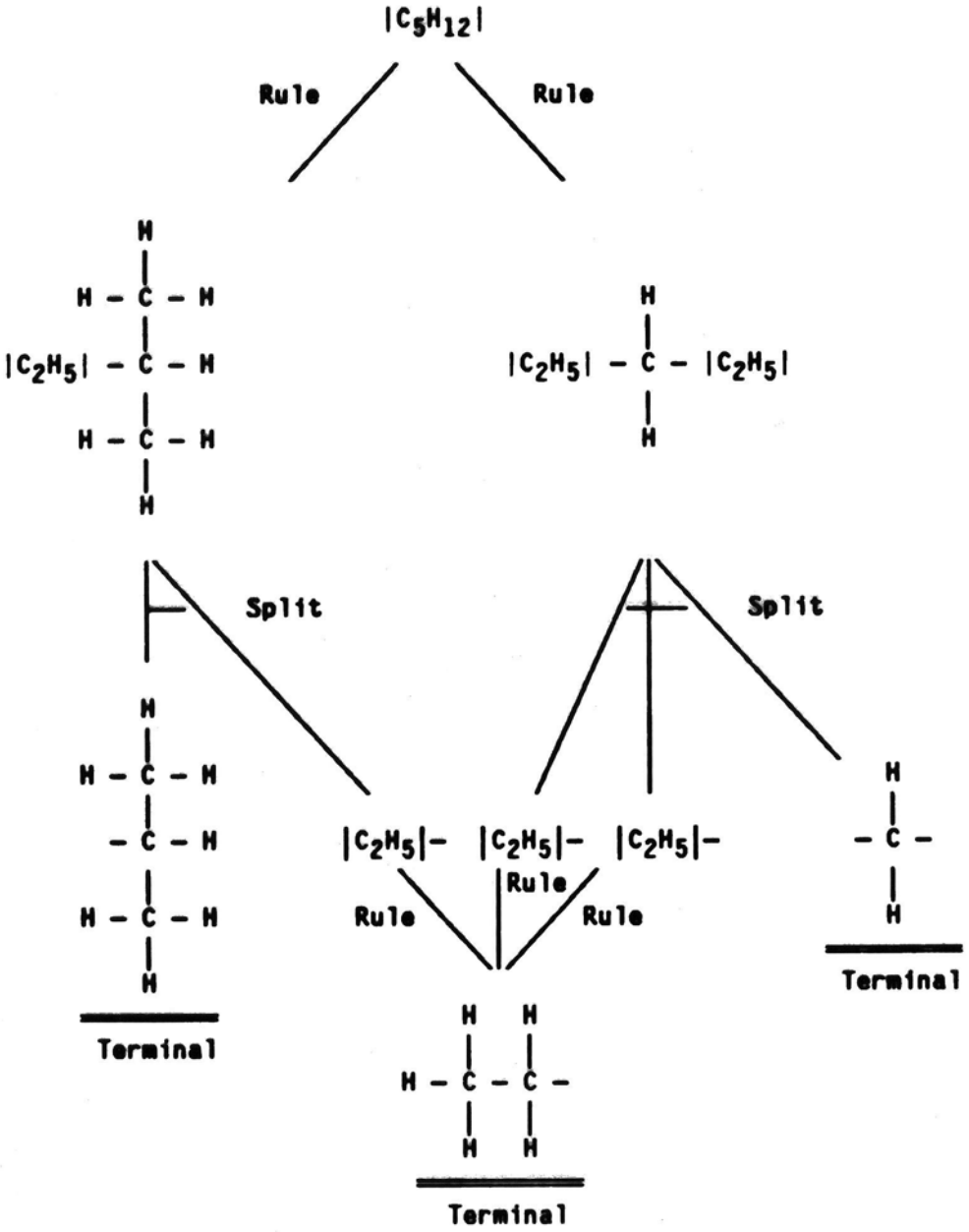
[3.3] penicillinase was added to this blood culture

then

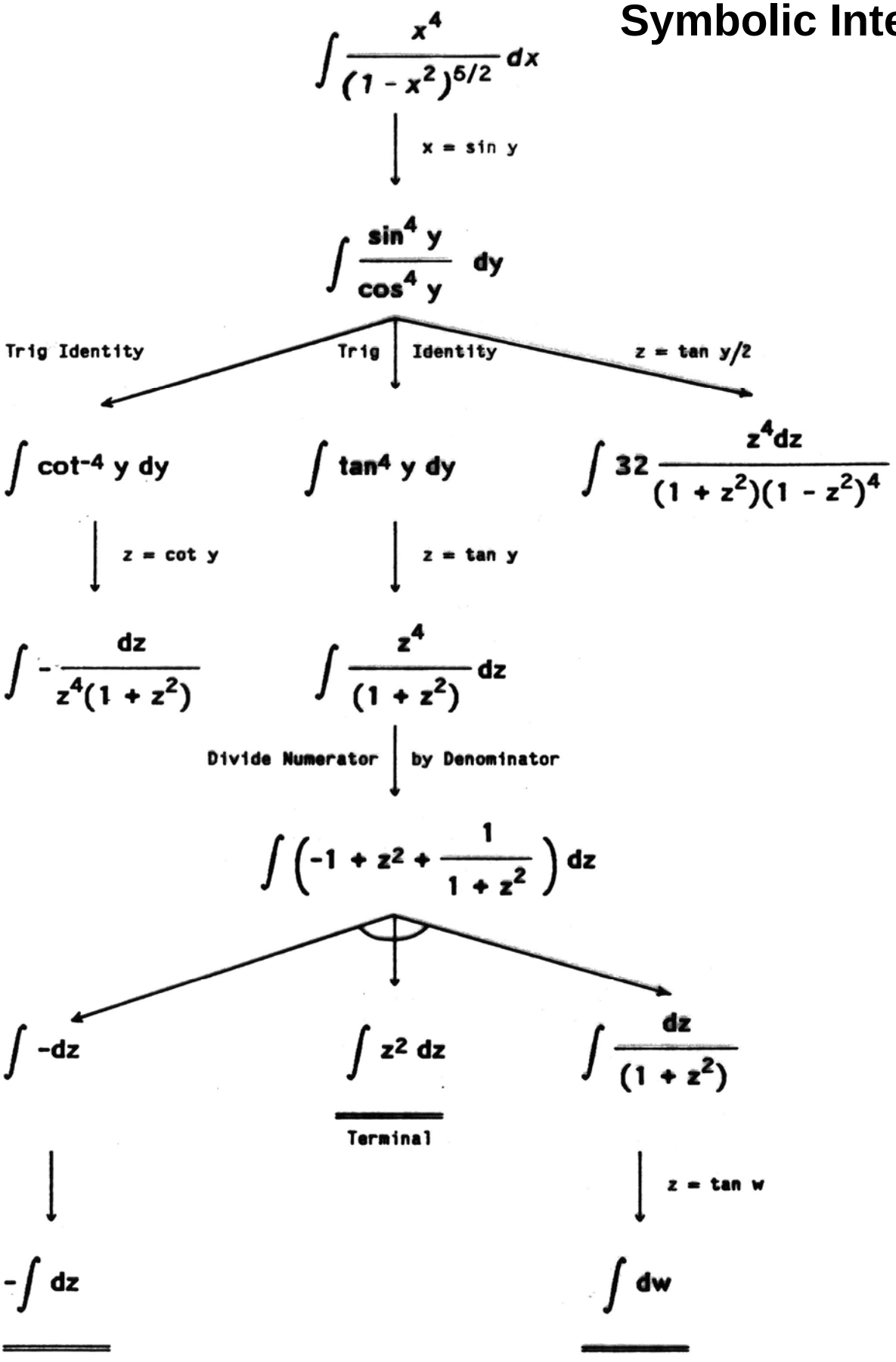
there is weakly suggestive evidence (.4) that ORGANISM-1 is a contaminant

[RULE039]

DENDRAL


|C5H12|
Rule
Rule
Split
Split
Rule
Rule
Rule
Terminal
Terminal
Terminal

Symbolic Integration

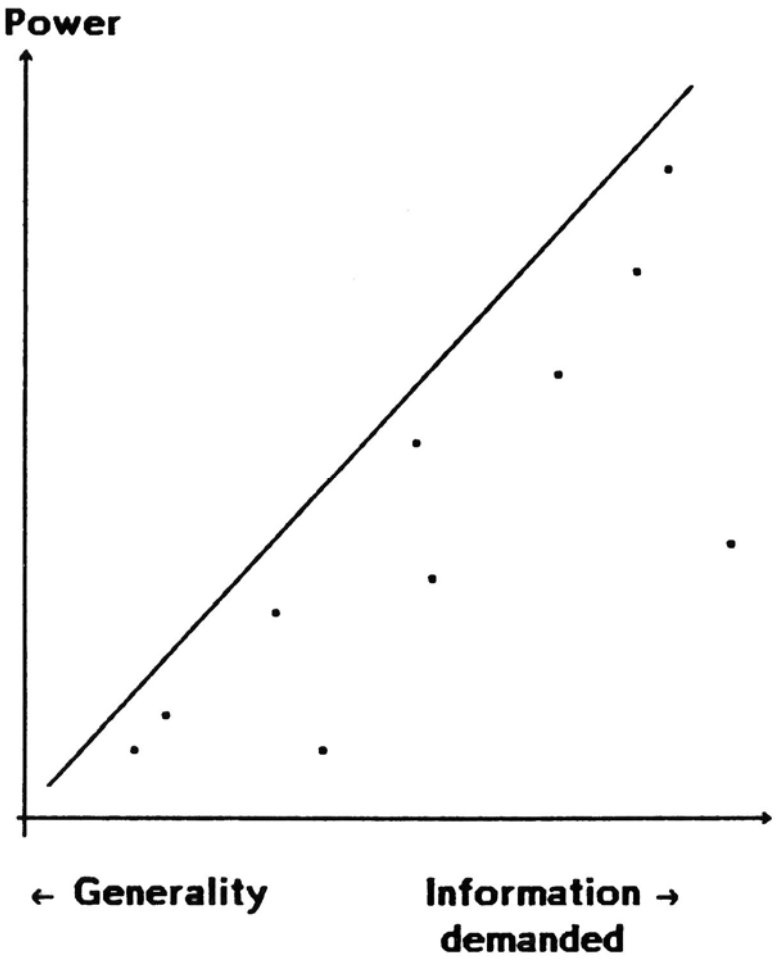


GPS: Power vs Generality

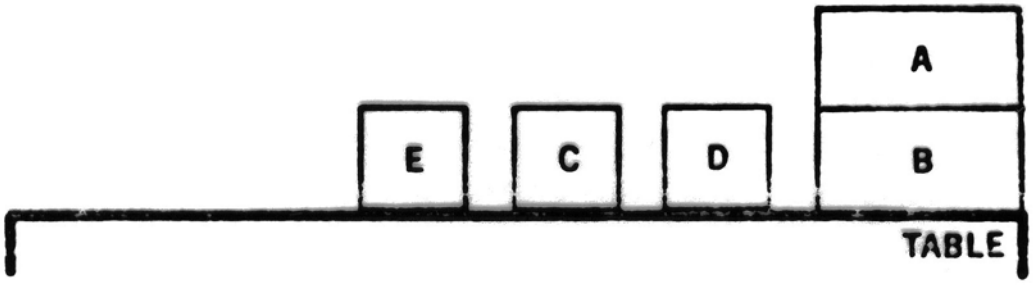
Generality:

- Missionaries and Cannibals
- Towers of Hanoi
- Father and Sons
- Three Coins Puzzle
- Bridges of Königsberg
- Letter Series Completion
- Symbolic Integration
- Simple Theorem-Proving
- Monkey and Bananas
- Parsing Sentences
- Water Jug Task

Failures: Increasing generality requires increased requirement on what must be given in problem statement.



PROBLEM: (MAKE (ON A B))



Automatic Programming

HACKER

Learns to construct simple programs in the “blocks world”

PECOS

Reachability Problem: Given a directed graph, G, and an initial vertex, v, find all vertices reachable from v by following zero or more arcs.

Robotics

Water Pump Assembly
visual and tactile sensing
Hinge Assembly
two arms
Industrial Automation
AL programming language

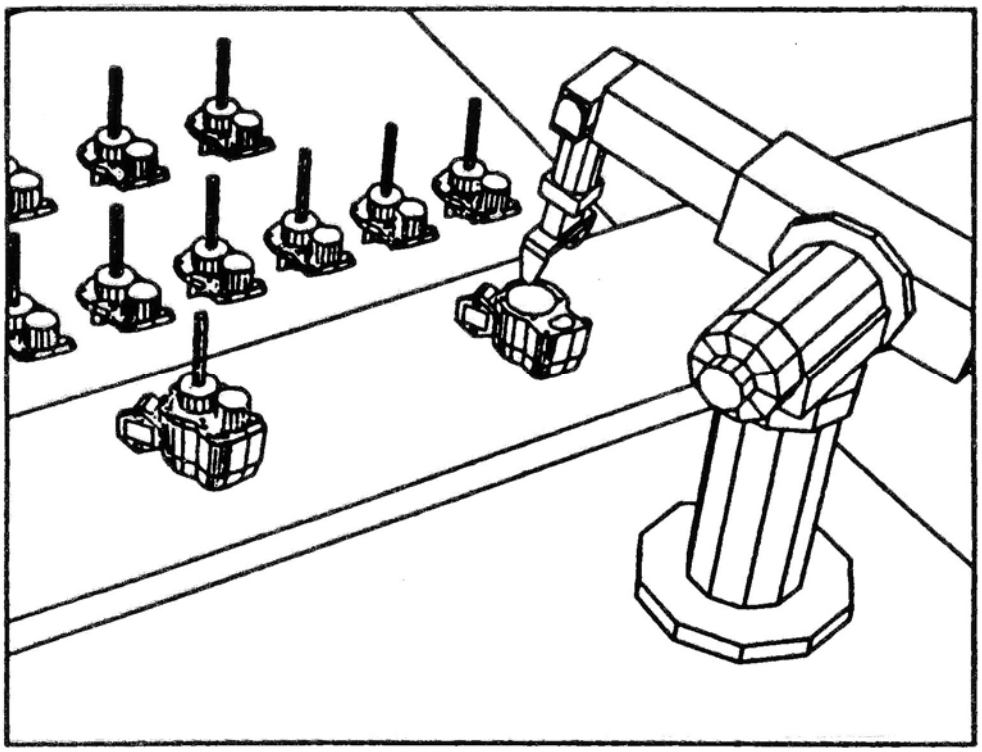
Machine Vision

Raw Images to Descriptions
Matching 3-D models to 2-D visual data
Blocks World (Roberts, Guzman, Waltz)
(illumination effects, shadows, occlusion)
COPY
uses a visual input device to look at a scene consisting of a structure of blocks. Then uses a robot arm to copy the structure from disarranged blocks.
Shape Description

Natural Language Systems

Machine Translation
Winograd (1972)
Shank
use of semantic primitives and scripts
Smart Database Systems

HEARSAY-II and HARPY (1000 words)
(commercial – 99% accuracy on 120 words)



Information Processing Psychology

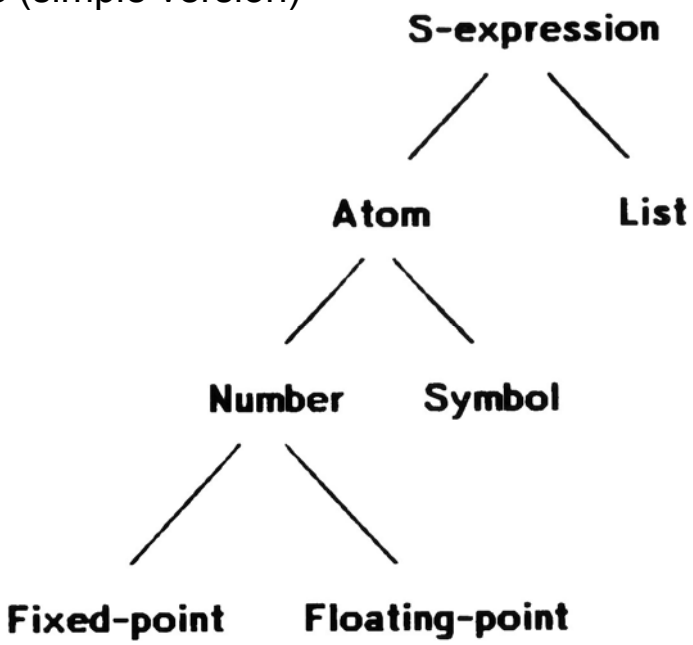
Concepts and vocabulary for useful theories of human behavior

EPAM
verbal learning

Newell and Simon
human problem solving

LISP Basics

Data Objects (simple version)



Most Lisp systems have other datatypes (e.g., strings, arrays, …).

Examples

```
THISISANATOM    T    NIL

1.5    7777Q

(A B C)    ( ) = NIL

((THIS IS AN EMBEDDED LIST) IN A LARGER LIST)

(S-expression
    (Atom
        (Number
            (Fixed-point Floating-point))
        (Symbol))
    (List))
```

Evaluation

Top Level of Lisp

```
WHILE TRUE DO
  BEGIN “top level”
    Read
    Eval
    Print
  END “top level”

PROCEDURE Eval
  BEGIN “eval”
    IF atom THEN look up value
    ELSE IF list THEN
      BEGIN “apply”
        Eval all list elements but
          the first and collect list
          of values
        Apply function definition of
          first element to resulting
          list of values and compute
          value
      END “apply”
    Return value
  END “eval”
```

Atom Values

1) An Atom can have any S-expression as its value, or have no value at all (be unbound – NOBIND).

2) Some Atoms are defined to *always* have a value.

Examples

```
Value-of-T = T
Value-of-NIL = NIL          always
Value-of-1 = 1
Value-of-3.14 = 3.14

Value-of-TERMINAL-TYPES = (DM2500 VT100)
```

Stopping Evaluation

If you type X at the top level of Lisp,
it will respond with: value-of-X.

Suppose you want to refer to X itself?

Precede X with ' (quote).

Thus, if you type 'X to the top level of Lisp,
it will respond with: X.

Lisp distinguishes between the *name* of an Atom and its *value*.

Notes

1) Every function returns a value and may have side effects (e.g., setting the value of an atom, printing something, …).

2) Eval doesn't always evaluate the remaining elements of the list before applying the function definition of the first element.

Setting Values

SETQQ SETQ SET

	Effect
`(SETQQ X Y)`	Value-of-X ← Y
`(SETQ X Y)`	Value-of-X ← Value-of-Y
`(SET X Y)`	Value-of-[?] ← Value-of-Y

Example

```
(SETQQ COMPUTER DEC-20)
    Value-of-COMPUTER = ?
    Value-of-DEC-20 = ?

(SETQQ LOAD-AVERAGE 7.5)
    Value-of-LOAD-AVERAGE = ?

(SETQ DEC-20 LOAD-AVERAGE)
    Value-of-DEC-20 = ?
    Value-of-COMPUTER = ?

(SETQ LOAD-AVERAGE 6.0)
    Value-of-LOAD-AVERAGE = ?
    Value-of-DEC-20 = ?

(SET COMPUTER LOAD-AVERAGE)
    Value-of-COMPUTER = ?
    Value-of-DEC-20 = ?

(SET DEC-20 7.5)     (SET 'DEC-20 7.5)

(SETQQ 'COMPUTER DEC-20)
```

Taking Lists Apart – 1

CAR returns the first element of a list.

Examples

```
(CAR ‘(A B C) = A

(CAR ‘((A B) C D E) = (A B)

(CAR X) = ?

(SETQQ X (A (B C)))

(CAR X) = A
```

CDR returns a list containing all but the first element.

Examples

```
(CDR ‘(A B C) = (B C)

(CDR ‘((A B) C D E) = (C D E)

(CDR X) = ((B C))
```

CAR and **CDR** have no side effects – just values.

The non-mnemonic names are carryovers (sigh!) from the original IBM 704 implementation: Contents of Address or Decrement Register.

Taking Lists Apart – 2

CDR and **CDR** can be composed for convenience.

```
(CAR (CDR X)) → (CADR X)

(CDR (CAR X)) → (CDAR X)

(CAR (CDR (CDR X))) → (CADDR X)
```

Examples

```
(CAAR '((A B) C D)) =
     (CAR (CAR '((A B) C D))) = A

(CADR '(A (B C) D)) = (B C)

(CDAR '((A B) C D)) = (B)

(CDDR '(A B C)) = (C)
```

All functions of the form C----R are defined as above (normally up to 4 in length – 8 for CDC).

Discrepancies: `(CAR NIL)` and `(CDR NIL)` are often defined as `NIL`.

Putting Them Back Together

CONS takes a list and inserts a new element. (**CONS** is a mnemonic for Constructor.

Examples

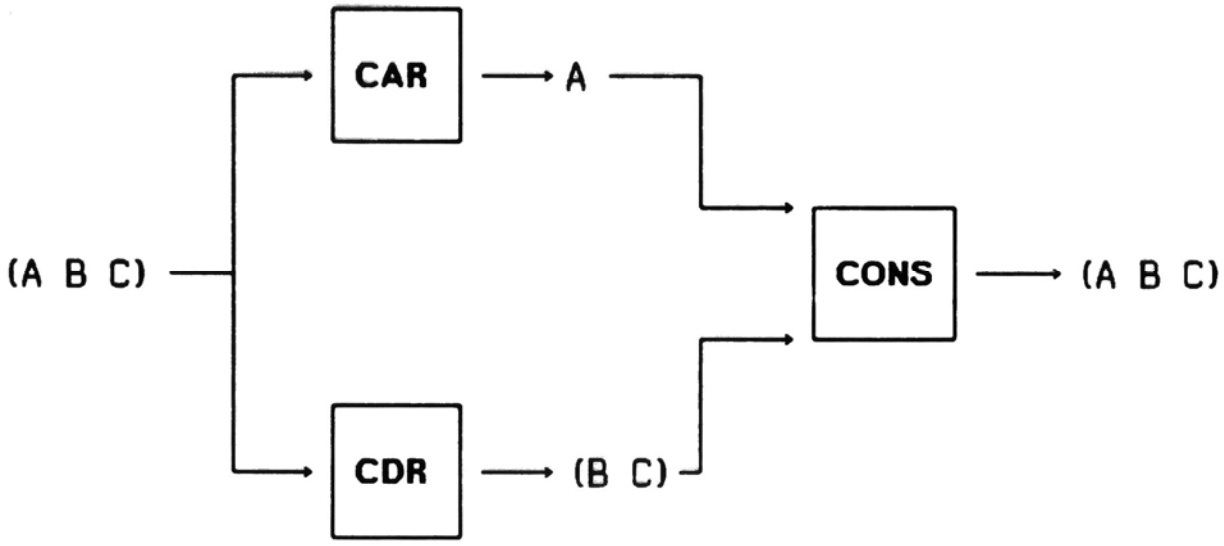
```
(CONS ‘A ‘(B C)) = (A B C)

(CONS ‘(A B) ‘(C D E)) = ((A B) C D E)

(CONS ‘A NIL) = (A)
```

CONS has a side effect as well as a value. It makes a new list.

CAR – CDR – CONS Summary



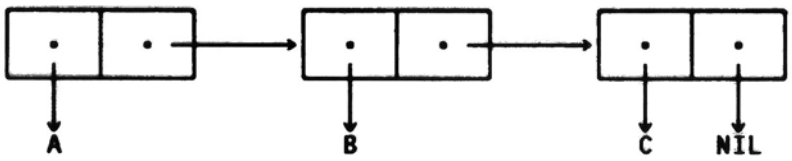

```
(CONS (CAR X) (CDR X))
```

List Storage

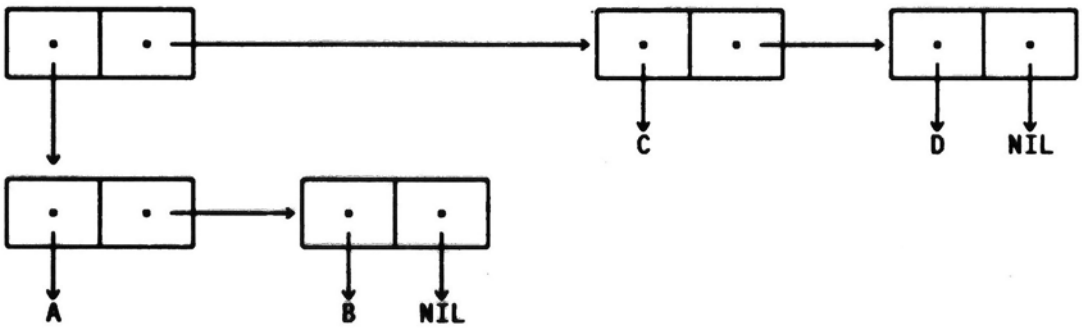
Lists are stored internally as a series of two-part cells, called CONS cells. Each cell contains two pointers – a CAR pointer and a CDR pointer.

Examples

(A B C)



((A B) C D)

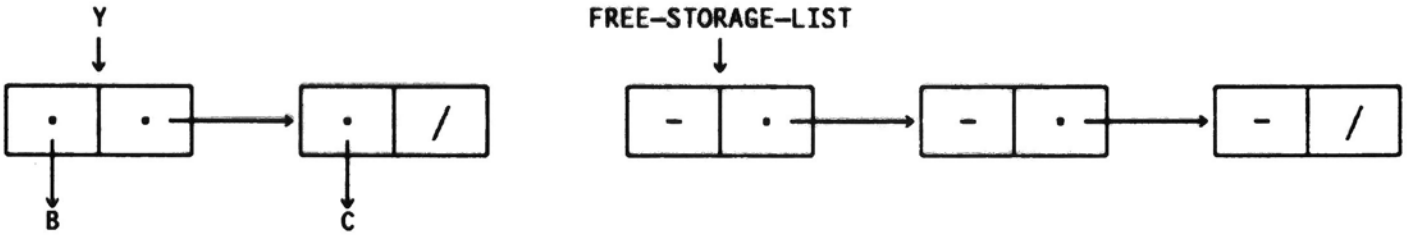


CONS Revisited

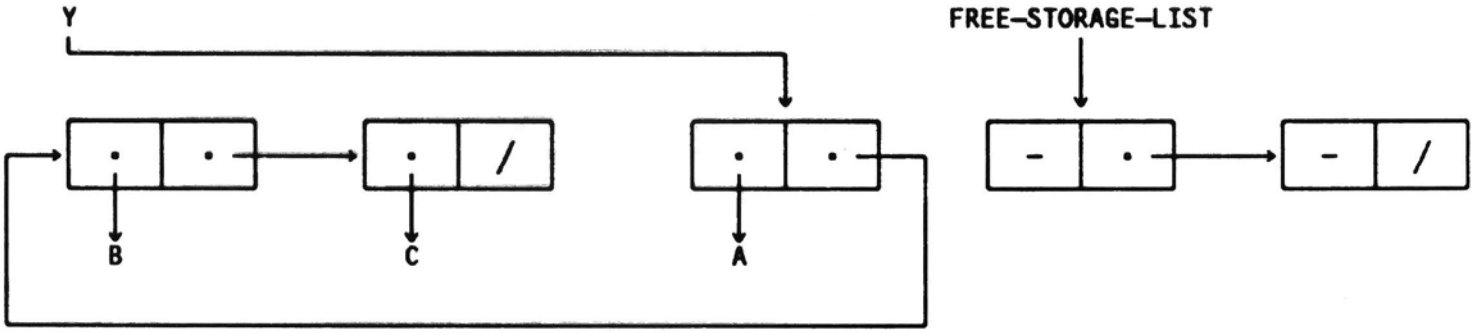
Lisp maintains a list of spare memory cells for its own use. This list is called the **Free Storage List. CONS** operates by removing the first cell on the Free Storage List and by depositing new pointers into this first cell.

Example

```
(SETQ Y '(B C))
```




```
(SETQ Y (CONS 'A Y))
```

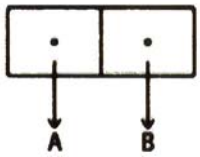


Dotted Pairs

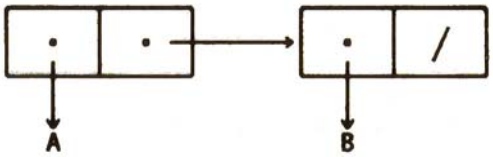
CONS doesn't *really* need a list as its second argument.

Example

```
(CONS 'A 'B)
```




```
(CONS 'A '(B))
```



This is called a Dotted Pair.

```
(CONS 'A 'B)      (A . B)
```

Dotted pairs are useful for conserving memory if it is really tight since `(A . B)` takes one memory cell less than `(A B)`. For our purposes, they can be ignored.

Garbage Collection

Consider this sequence:

```
(SETQ Y (CONS ‘A (B C)))

(SETQ Y (CONS ‘X (Y Z)))
```

After the first SETQ, the value of Y is?

After the second SETQ, the value of Y is?

What happened to `(A B C)`?

It is no longer accessible – the cells that it uses are wasted to the rest of the system.

Garbage Collection is the process by which inaccessible cells are returned to the Free Storage List. It is often done in two phases: a *mark* phase and a *sweep* phase. In the first phase, the garbage collector runs through memory and marks all accessible cells. In the second phase, the collector runs through memory again and adds all inaccessible cells to the Free Storage List.

Testing for Equality

EQUAL takes two arguments and returns T if they are the same.

```
(EQUAL ‘Y ‘Y) = T

(EQUAL ‘(A B C) ‘(A B C)) = T

(EQUAL ‘(A B C) ‘((A B C))) = NIL
```

EQ takes two arguments and returns T if they are the same atom. However, if the arguments are lists, then **EQ** returns T if an only if they are represented by exactly the same memory structure.

```
(EQ ‘Y ‘Y) = T

(SETQ X ‘(A B C))

(SETQ Y ‘(A B C))

(EQ X Y) = NIL  but  (EQUAL X Y) = T

(SETQ YY Y)

(EQ YY Y) = T        (EQUAL YY Y) = T
```

Why use **EQ**? A matter of efficiency. It takes longer to see if a structure is a copy than to see if it is exactly the same. Beginners are advised to avoid **EQ.**

Some Useful Functions

LENGTH returns the number of elements in a list.

```
(LENGTH ‘(A B C)) = 3

(LENGTH ‘((A B) C (D E ((F))))) = ?
```

LIST makes a list out of its arguments.

```
(LIST ‘A ‘B ‘C) = (A B C)

(LIST ‘(A B) ‘(C D)) = ((A B) (C D))
```

APPEND strings together the elements of all lists supplied as arguments.

```
(APPEND ‘(A) ‘(B C)) = (A B C)

(APPEND ‘(A (B)) ‘(C D)) = (A (B) C D)

(APPEND ‘(A) ‘() ‘(B) ‘()) = (A B)
```

APPEND actually copies the first list and attaches the second list to the copy. (Generalize this to *n* arguments.)

REVERSE reverses the order of the elements of a list.

```
(REVERSE ‘(A B C)) = (C B A)

(REVERSE ‘((A B) ‘(C D))) = ((C D) (A B))
```

Predicates

A **predicate** is a function that returns T or NIL, where T and NIL correspond to logical values of true and false. (Predicates are often implemented so that they return NIL if false, but instead of T if true, some other useful value.)

ATOM returns T if its argument is an atom.

```
(ATOM ‘X) = T

(ATOM 6.5) = T

(ATOM ‘(A B C)) = NIL
```

NULL returns T if its argument is the empty list.

```
(NULL ‘()) = (NULL NIL) = T

(NULL T) = NIL
```

MEMBER tests to see if its first argument is an element of its second argument (a list). If true, **MEMBER** returns the fragment of the list that begins with the first argument.

```
(MEMBER ‘A ‘(A B C)) = (A B C)

(MEMBER ‘B ‘(A B C)) = (B C)

(MEMBER ‘Y ‘(A B C)) = NIL

(MEMBER ‘(A (B)) ‘(C (A (B)) G)) = ((A (B)) G)
```

Arithmetic

NUMBERP returns NIL if its argument is not a number. Otherwise it returns the number.

```
(NUMBERP 0) = 0  (NUMBERP -6.5) = -6.5

(NUMBERP ‘Y) = NIL
```

ZEROP returns T if its argument is 0.

LESSP returns T if its first argument is less than its second argument.

GREATERP returns T if its first argument is greater than its second argument.

```
(GREATERP 7.3 6.1) = T    (LESSP 7.3 6.1 = NIL
```

PLUS adds its arguments together.

TIMES multiplies its arguments together.

DIFFERENCE subtracts its second argument from its first argument.

QUOTIENT divides its first argument by its second argument.

ADD1 adds 1 to its argument. **SUB1** subtracts 1 from its argument.

```
(PLUS 1 2 3 -4 5) = 7     (TIMES 1 2 3 -4 5) = -120

(DIFFERENCE 1 3) = -2     (QUOTIENT 5.0 2.5) = 2.0

(QUOTIENT 1 2) = 0    (ADD1 2) = 3    (SUB1 1) = 0
```

List-Altering Functions

RPLACA replaces the CAR part of a CONS cell with a new pointer.

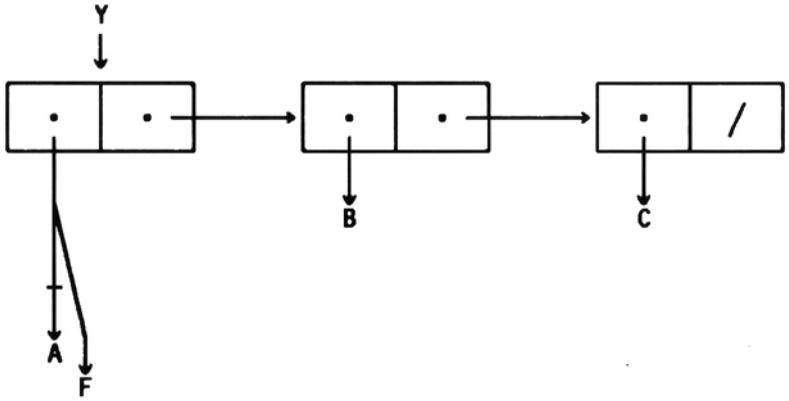
RPLACD replaces the CDR part of a CONS cell with a new pointer.

NCONC is like APPEND, but it makes no copies.

Examples

```
(SETQ Y '(A B C)
```

```
(RPLACA Y 'F)
```




```
(RPLACD Y '(D E))          Y = ?

(SETQ X '(A B C))          (SETQ Y '(D E F))

(SETQ YY '(APPEND X Y))    YY = ?  X = ?  Y = ?

(SETQ YY '(NCONC X Y))     YY = ?  X = ?  Y = ?
```

COND

COND is the LISP conditional.

```
(COND (<test 1> <form> <form> ... <form 1>)
      (<test 2> <form> <form> ... <form 2>)
......(<test 3> <form> <form> ... <form 3>)
      .
      .
      .
      (<test n> <form> <form> ... <form n>))
```

Each list is called a *clause*. **COND** searches through the clauses evaluating only the first element of each until one is found whose value is non-NIL. Then everything in the successful clause is evaluated and the last thing evaluated is returned as the value of the **COND**. (The intervening forms are therefore evaluated only for their side effects.) If no successful clause is found, the **COND** returns NIL. If the successful clause consists of only one element, then the value of that element is returned.

Example

```
(COND ((LESSP X Y) Y)
      (T X))
```

SELECTQ

SELECTQ is a useful form of CASE function.

```
(SELECTQ PATT (<patt 1> <form> <form> ... <form 1>)
              (<patt 2> <form> <form> ... <form 2>)
              .
              .
              .
              (<patt n> <form> <form> ... <form n>)
              <default>)
```

AND / OR

AND returns T if all of its arguments have non-NIL values. Evaluation of the arguments proceeds left-to-right and stops with the first argument to have value NIL.

OR returns T if at least one of its arguments has a non-NIL value. Evaluation of the arguments proceeds left-to-right and stops with the first argument to have a non-NIL value.

Example

```
(AND NEEDTODOTHIS (SETQ X (CAR Y)))

(OR ALREADYDIDTHIS (SETQ X (CAR Y)))
```

Defining Functions

```
(DEF (<function-name> <argument list>
      <form 1> <form 2> ... <form n>))
```

Example

```
(DEF (AUGMENT (ITEM BAG)
          (COND ((MEMBER ITEM BAG) BAG)
                (T (CONS ITEM BAG)))))

(DEFINEQ (AUGMENT (LAMBDA (ITEM BAG)
                    (COND ((MEMBER ITEM BAG) BAG)
                          (T (CONS ITEM BAG))))))
```

Variable Binding

Variables can be bound or free.

A *bound* variable, with respect to a function, is an atom that appears in the function's parameter list.

A *free* variable, with respect to a function, is an atom that does not appear in the function's parameter list.

It makes no sense to speak of a variable as bound or free unless we also specify with respect to what function the variable is bound or free.

Bound variable values must be saved, so that they may be restored. If a bound variable is also used as a free variable, its value is the current one, not any that may have been saved.

Example

```
(DEF (INCREMENT (PARAMETER)
          (SETQ PARAMETER (PLUS PARAMETER FREE))
          (SETQ OUTPUT PARAMETER)))

(SETQ PARAMETER 15)     (SETQ FREE 10)

(SETQ OUTPUT 10)        (SETQ ARGUMENT 10)

(INCREMENT ARGUMENT) = ?

OUTPUT = ?    PARAMETER = ?    ARGUMENT = ?
```

Variable Binding

Lisp is neither Call-by-Reference nor Call-by-Value.

When the argument to a function is an atom, the function's parameter is *bound* to the value of the atomic argument. There is no other choice.

Since there is no copying involved, it sounds like call-by-reference. Except … the value of the argument is not usually altered by changes to the value of the parameter. It sounds like call-by-value. Except … destructive functions like RPLACA can change the value of the argument when they are used on the parameter.

Conclusion: What Lisp does corresponds most closely to call-by-reference, but for most purposes it can be thought of as using call-by-value. The only time is makes a difference is when list structure altering functions are used.

Free-variable values are determined *dynamically*, not *lexically*.

Evaluation Environment: The environment (i.e., collection of bindings) in force when the function requiring the free-variable values is evaluated (*dynamic scoping*).

Definition Environment: The environment in force when the function requiring the free-variable values is defined (*lexical scoping*).

Lambda Definitions

LAMBDA defines anonymous functions.

Functions are represented internally as LAMBDA expressions.

Example

```
(DEF (AUGMENT (ITEM BAG)
          (COND ((MEMBER ITEM BAG) BAG)
                (T (CONS ITEM BAG)))))
```

results in the function definition.

```
AUGMENT:
    (LAMBDA (ITEM BAG)
            (COND ((MEMBER ITEM BAG) BAG)
                  (T (CONS ITEM BAG))))
```

We will see that it is sometimes appropriate to define functions that do not have names – typically for functions that are needed in a particular situation but aren't generally useful – so we don't clog the system with extra names.

Recursion

Example

```
(MEMBER
  (LAMBDA (ITEM S)
    (COND
      ((NULL S) NIL)
      ((EQUAL ITEM (CAR S)) S)
      (T (MEMBER ITEM (CDR S))))))

(MEMBER 'C '(A B C))

MEMBER: ITEM = C    S = (A B C)
   MEMBER: ITEM = C    S = (B C)
      MEMBER: ITEM = C    S = (C)
      MEMBER = (C)
   MEMBER = (C)
MEMBER = (C)
```

A function is *tail recursive* if values are passed upward without alteration as the recursion unwinds.

Recursion

Example

```
(OCC
  (LAMBDA (AT S)
    (COND
      ((ATOM S) (EQ AT S))
      (T (OR (OCC AT (CAR S))
             (OCC AT (CDR S)))))))

(OCC 'A '(P (G A)))

OCC: AT = A    S = (P (G A))
   OCC: AT = A    S = P
   OCC = NIL
   OCC: AT = A    S = ((G A))
      OCC: AT = A    S = (G A)
         OCC: AT = A    S = G
         OCC = NIL
         OCC: AT = A    S = (A)
            OCC: AT = A    S = A
            OCC = T
         OCC = T
      OCC = T
   OCC = T
OCC = T
```

Recursion

Example

```
(EQUAL
  (LAMBDA (X Y)
    (COND
      ((ATOM X) (EQ X Y))
      ((ATOM Y) NIL)
      (T (AND (EQUAL (CAR X) (CAR Y))
              (EQUAL (CDR X) (CDR Y)))))))

(EQUAL ‘(A B C) ‘(A B C))

EQUAL: X = (A B C)    Y = (A B C)
   EQUAL: X = A    Y = A
   EQUAL = T
   EQUAL: X = (B C)    Y = (B C)
      EQUAL: X = B    Y = B
      EQUAL = T
      EQUAL: X = (C)    Y = (C)
         EQUAL: X = C    Y = C
         EQUAL = T
         EQUAL: X = NIL    Y = NIL
         EQUAL = T
      EQUAL = T
   EQUAL = T
EQUAL = T
```

Recursion

Example

```
(POWER
  (LAMBDA (M N)
    (COND
      ((ZEROP N) 1)
      (T (TIMES M (POWER M (SUB1 N)))))))

(POWER 2 3)

POWER: M = 2    N = 3
   POWER: M = 2    N = 2
      POWER: M = 2    N = 1
         POWER: M = 2    N = 0
         POWER = 1
      POWER = 2
   POWER = 4
POWER = 8
```

Writing Recursive Functions

To find the value for an arbitrary argument, ask for what simpler arguments the value of the function must be known.

1) Determine trivial argument values and termination conditions.

2) Determine the method of decomposition for an arbitrary argument.

3) Determine the method for constructing the result.

Recursion Schemata

1) **List Recursion:** The value of the function is immediate for the argument NIL. Otherwise, it depends only on the value for the CDR of the argument (e.g., MEMBER).

2) **S-expression Recursion:** The value of the function is immediate for atomic values of the argument. Otherwise, it depends on the values for the CAR and the CDR of the argument (e.g., EQUAL).

3) **Other Structural Recursions:** When a list is used to represent an algebraic expression, functions of this expression often have a recursive form closely related to its inductive definition (e.g., POWER).

Using Auxiliary Functions

For some problems, an auxiliary function with an extra argument can be used to accumulate a partial result. This is often simpler than trying to solve the problem with a single function.

Example

```
(REVERSE
  (LAMBDA (X) (REV X NIL)))

(REV
  (LAMBDA (X Y)
    (COND
      ((NULL X) Y)
      (T (REV (CDR X) (CONS (CAR X) Y))))))

(REVERSE '(A B C))

REVERSE: X = (A B C)
   REV: X = (A B C)    Y = NIL
      REV: X = (B C)    Y = (A)
         REV: X = (C)    Y = (B A)
            REV: X = NIL    Y = (C B A)
            REV = (C B A)
         REV = (C B A)
      REV = (C B A)
   REV = (C B A)
REVERSE = (C B A)
```

Some Lisp implementations allow you to pad functions with extra arguments to obviate the need for auxiliary functions.

Iteration

PROG creates new variables and provides a way to write functions that iterate.

Example

```
(POWER
  (LAMBDA (M N)
    (PROG (RESULT EXPONENT)
          (SETQ RESULT 1)
          (SETQ EXPONENT N)
     LOOP (COND
            ((ZEROP EXPONENT) (RETURN RESULT)))
          (SETQ RESULT (TIMES M RESULT))
          (SETQ EXPONENT (SUB1 EXPONENT))
          (GO LOOP))))
```

The first position after PROG is occupied by a list of parameters that are bound (to NIL) on entering the PROG and restored to old values on exit.

The value of a PROG is the value of the RETURN that stops it. A PROG that drops off the end has a NIL value.

Please don’t use PROG to write FORTRAN code in Lisp!

MAPCAR: List Iteration

A pattern that occurs over and over in writing Lisp functions is the following list recursion.

```
(fLIST
  (LAMBDA (L)
    (COND
      ((NULL L) NIL)
      (T (CONS (f (CAR L))
               (fLIST (CDR L))))))
```

Example

```
(DROP
  (LAMBDA (L)
    (COND
      ((NULL L) NIL)
      (T (CONS (LIST (CAR L))
               (DROP (CDR L))))))

(DROP '(A B C))

DROP: L = (A B C)
   DROP: L = (B C)
      DROP: L = (C)
         DROP: L = NIL
         DROP = NIL
      DROP = ((C))
   DROP = ((B) (C))
DROP = ((A) (B) (C))
```

Lisp allows functional arguments to be passed. This enables us to simplify such recursions using the function **MAPCAR.**

MAPCAR: List Iteration

```
(MAPCAR
  (LAMBDA (L F)
    (COND
      ((NULL L) NIL)
      (T (CONS (F (CAR L))
               (MAPCAR (CDR L) F))))))
```

Example

```
(DROP
  (LAMBDA (L)
    (MAPCAR L (FUNCTION LIST))))

(DROP '(A B C))

DROP: L = (A B C)
   MAPCAR: L = (A B C)    F = LIST
   MAPCAR = ((A) (B) (C))
DROP = ((A) (B) (C))
```

DROP has the following iterative translation

```
(DROP
  (LAMBDA (L)
    (for X in L collect (LIST X))))
```

MAPCAR and APPLY

APPLY applies a function to a list of arguments. (It is the function used by EVAL.) MAPCAR and APPLY make an effective combination.

Consider the following:

```
(ATOMS
  (LAMBDA (S)
    (COND
      ((NULL S) 0)
      ((ATOM S) 1)
      (T (PLUS (ATOMS (CAR S))
               (ATOMS (CDR S)))))))
```

MAPCAR and APPLY

```
(ATOMS '(TIMES X (SQRT 4)))

ATOMS: S = (TIMES X (SQRT 4))
   ATOMS: S = TIMES
   ATOMS = 1
   ATOMS: S = (X (SQRT 4))
      ATOMS: S = X
      ATOMS = 1
      ATOMS: S = ((SQRT 4))
         ATOMS: S = (SQRT 4)
            ATOMS: S = SQRT
            ATOMS = 1
            ATOMS: S = (4)
               ATOMS: S = 4
               ATOMS = 1
               ATOMS: S = NIL
               ATOMS = 0
               PLUS: 1 0
               PLUS = 1
            ATOMS = 1
            PLUS: 1 1
            PLUS = 2
         ATOMS = 2
         ATOMS: S = NIL
         ATOMS = 0
         PLUS: 2 0
         PLUS = 2
      ATOMS = 2
      PLUS: 1 2
      PLUS = 3
   ATOMS = 3
   PLUS: 1 3
   PLUS = 4
ATOMS = 4
```

MAPCAR and APPLY

Rewrite ATOMS.

```
(ATOMS
  (LAMBDA (S)
    (COND
      ((NULL S) 0)
      ((ATOM S) 1)
      (T (APPLY (FUNCTION PLUS)
                (MAPCAR S (FUNCTION ATOMS)))))))

(ATOMS '(TIMES X (SQRT 4)))

ATOMS: S = (TIMES X (SQRT 4))
   MAPCAR: L = (TIMES X (SQRT 4))    F = ATOMS
      ATOMS: S = TIMES
      ATOMS = 1
      ATOMS: S = X
      ATOMS = 1
      ATOMS: S = (SQRT 4)
         MAPCAR: L = (SQRT 4)        F = ATOMS
            ATOMS: S = SQRT
            ATOMS = 1
            ATOMS: S = 4
            ATOMS = 1
         MAPCAR = (1 1)
         PLUS: 1 1
         PLUS = 2
      ATOMS = 2
   MAPCAR = (1 1 2)
   PLUS: 1 1 2
   PLUS = 4
ATOMS = 4
```

Unfortunately, some Lisp implementations don't allow you to APPLY functions that have an indefinite number of arguments (like PLUS).

MAPCAR revisited

In some Lisp implementations, MAPCAR must be written like this (e.g., INTERLISP).

```
(MAPCAR
  (LAMBDA (L F)
    (COND
      ((NULL L) NIL)
      (T (CONS (APPLY F (LIST (CAR L)))
               (MAPCAR (CDR L) F))))))
```

Atom Properties and Association Lists

Atoms can have any number of properties as well as a value.

PUT stores a property and value on the property list of an atom.

GET retrieves a property value from the property list of an atom.

Examples

```
(PUT ‘PHEASANT ‘IS-A ‘BIRD)

(GET ‘PHEASANT ‘IS-A)
```

An Association List (A-list) is a list of pairs.

Example

```
(SETQ STATUS
    ‘((TEMPERATURE 103) (PRESSURE 120 60) (PULSE 72)))
```

ASSOC (SASSOC for CYBER) searches its second argument (an a-list) for a pair whose CAR is equal to its first argument (the *key*). It returns the pair so discovered, else NIL.

Examples

```
(ASSOC ‘PRESSURE STATUS) = (PRESSURE 120 60)

(ASSOC ‘COMPLAINTS STATUS) = NIL
```

Production Systems

1. **Global Database**

2. **Set of Production Rules**
   Precondition → Action

3. **Control System**
   Rule Application
   Termination Condition

The *Representation Problem*: How to transform a problem statement into the above three components.

Example

Missionaries and Cannibals Problem: You have N missionaries and N cannibals on the left bank of a river. You also have a boat that will carry (N + 1)/2 persons. Your problem is to transport all missionaries and cannibals to the right bank of the river in such a way that there are never more cannibals than missionaries in any one spot.

Consider N = 3, so the boat carries 2 persons.

States Moves Goal

Problem Space

Production Systems

State Vector Representation:

$(M_L \quad C_L \quad B)$

M_L = Number of missionaries on left bank

C_L = Number of cannibals on left bank

B = Boat Position (L = left, R = right)

Initial State: (3 3 L)

Goal State: (0 0 R)

N = Number of missionaries and number of cannibals

Production Rules

Move Left-to-Right (m,c)

Precondition:

$B = L$

$m \geq c$

$(M_L - m \geq C_L - c) \vee (M_L - m = 0)$

$(N - (M_L - m) \geq N - (C_L - c)) \vee ((N - M_L - m) = 0)$

Action:

$B \leftarrow R$

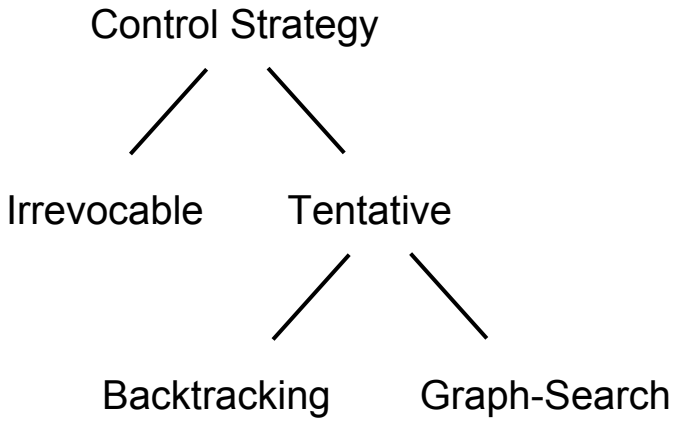
$M_L \leftarrow M_L - m$

$C_L \leftarrow C_L - c$

Production Systems

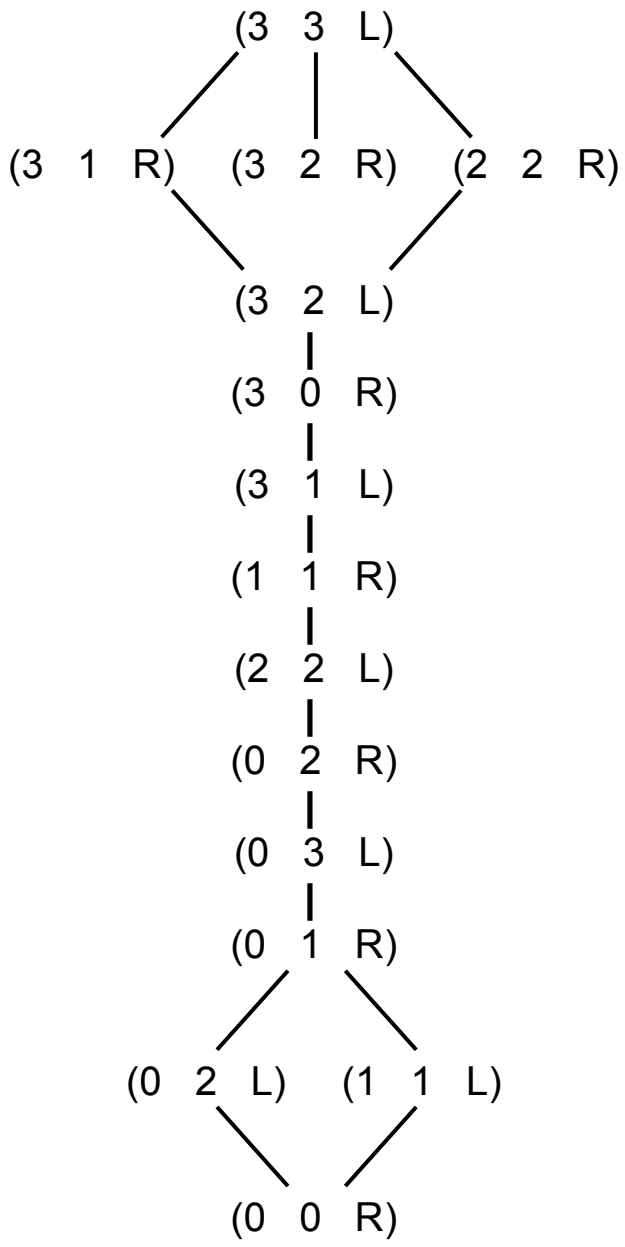
Control: A *nondeterministic* algorithm

```
PROCEDURE Production
  BEGIN "production"
    DATA ← Initial Database
    UNTIL DATA satisfies termination condition, DO
      BEGIN "apply rule"
      RULE ← SELECT some rule in the set of rules
                  that can be applied to DATA
      DATA ← Result of applying RULE to DATA
      END "apply rule"
  END "production"
```



Production Systems

Missionaries and Cannibals Solution Trace



Specialized Production Systems

Commutative Production Systems

a) For any database, *d*, if rule *r* applies to *d*, then r applies to any descendant of *d*.

b) If *d* satisfies the goal condition, then any database produced by applying *r* to *d* also satisfies the goal condition.

c) Given a set *R* of rules applicable to *d*, the result of applying each member of *R* in sequence to *d* is independent of the order of the sequence.

This is interesting because:

a) An irrevocable control scheme is acceptable (assured of finding a solution).

b) Once a rule has been found to be applicable, its preconditions no longer need to be tested.

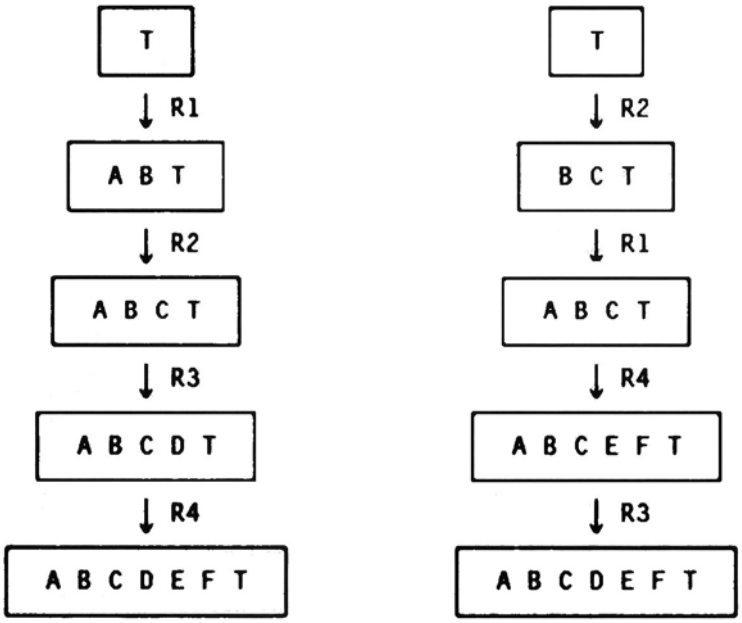
Specialized Production Systems

Example: Theorem-proving

Database: T

Rules:

R1: T → A, B
R2: T → B, C
R3: A → D
R4: B → E, F



Specialized Production Systems

Decomposable Production Systems

There exists a way to decompose the database so that parts of it can be processed independently, and so that the separate results can be combined into the solution of the original problem.

Must be able to decompose both the database and the termination condition.

This is interesting because:

a) It allows parallel processing.

b) Even without parallel processing it allows more efficient computation (by avoiding exploration of redundant paths).

If the content of the database represents a problem, the decomposition results in series of subproblems to be solved. This approach is called **Problem Reduction.** (Note that the decomposition is recursive.)

Specialized Production Systems

```
PROCEDURE Split
  BEGIN “split”
    DATA ← Initial Database
    {Di} ← Decomposition of DATA; the individual Di
           are now regarded as separate databases
    {Ti} ← Decomposition of termination condition
    UNTIL each Di satisfies Ti, DO
      BEGIN “apply rule”
        D* ← SELECT some database from {Di} that
                    does not satisfy Ti
        Remove D* from {Di}
        RULE ← SELECT some rule in the set of rules
                      that can be applied to D*
        D* ← Result of applying RULE to D*
        {di} ← Decomposition of D
        Append {di} to {Di}
      END “apply rule”
  END “split”
```

Forward and Backward Reasoning

Forward Reasoning:

Working from the initial state to the goal state
Working from known facts to new, deduced facts
Forward Chaining / Data-driven Reasoning
Bottom-up Reasoning / Antecedent Reasoning

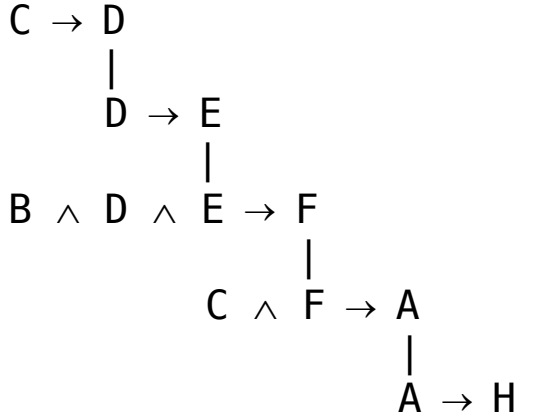
Backward Reasoning:

Working from the goal state to the initial state
Working from hypothesized conclusions back to known facts
Backward Chaining / Goal-directed Reasoning
Top-down Reasoning / Consequent Reasoning

Example: Try to deduce that H is true.

Database: B C

Rules: R1. $B \wedge D \wedge E \rightarrow F$
R2. $D \wedge G \rightarrow F$
R3. $C \wedge F \rightarrow A$
R4. $C \rightarrow D$
R5. $D \rightarrow E$
R6. $A \rightarrow H$



Graph Notation

Directed Graph

Nodes and *Arcs*
Arcs are *directed* from one node to another
Nodes = Databases / Arcs = Rules
Given an arc directed from node n_i to node n_j, n_j is called a *successor* of n_i, and n_i is a parent of n_j
A sequence of nodes ($n_{i,1}$, $n_{i,2}$, …, $n_{i,k}$), with each $n_{i,j}$ a successor of $n_{i,j-1}$ is called a path of length *k* from $n_{i,1}$ to $n_{i,k}$
n_j is *accessible* from n_i if a path exists. n_j is a *descendant* of n_i and n_i is an *ancestor* of n_j

Tree

A node has at most one parent
A node with no parent is a *root* node
A node with no successors is a *tip* node
The root node has depth 0
The depth of any other node is the depth of its parent node plus 1

Cost

$c(n_i,n_j)$ is the cost of the path from n_i to n_j (i.e., the cost of applying the rules)

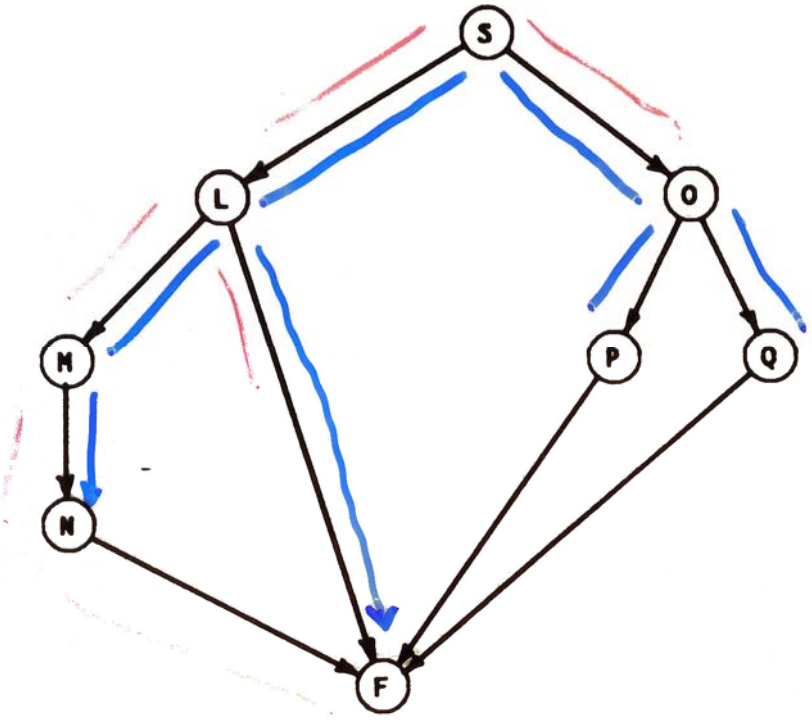
Problem: Find a path form node *s* (the initial database) to goal node *t* (the goal database), or to any one of a set of nodes $\{t_i\}$ that satisfies the termination condition

Explicit vs *Implicit* Graphs

Graph Search

```
(GRAPH-SEARCH
  (LAMBDA (s)
     (PROG (OPEN CLOSED n M)
           (SETQ OPEN (LIST s))
           (SETQ CLOSED NIL)
      LOOP (COND
             ((NULL OPEN)
               (RETURN 'FAIL)))
           (SETQ n (CAR OPEN))
           (SETQ OPEN (CDR OPEN))
           (SETQ CLOSED (CONS n CLOSED))
           (COND
             ((GOAL-NODE n)
               (RETURN (TRACE-PATH n s))))
           (SETQ M (EXPAND n))
           (SETQ M (POINTERS M n OPEN CLOSED))
           (SETQ OPEN (ORDER M OPEN))
           (GO LOOP))))
```

Search Strategies

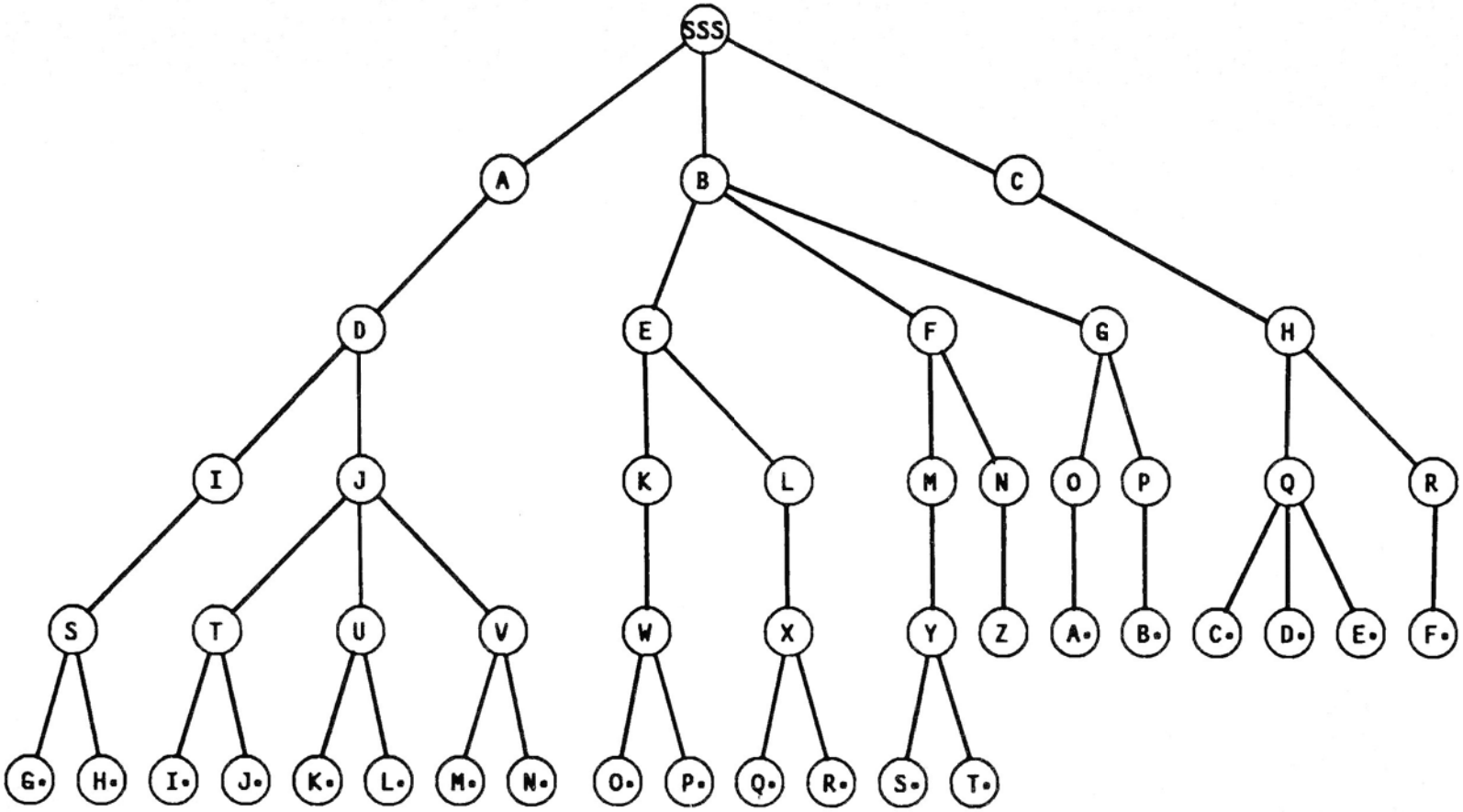


Uninformed Strategies

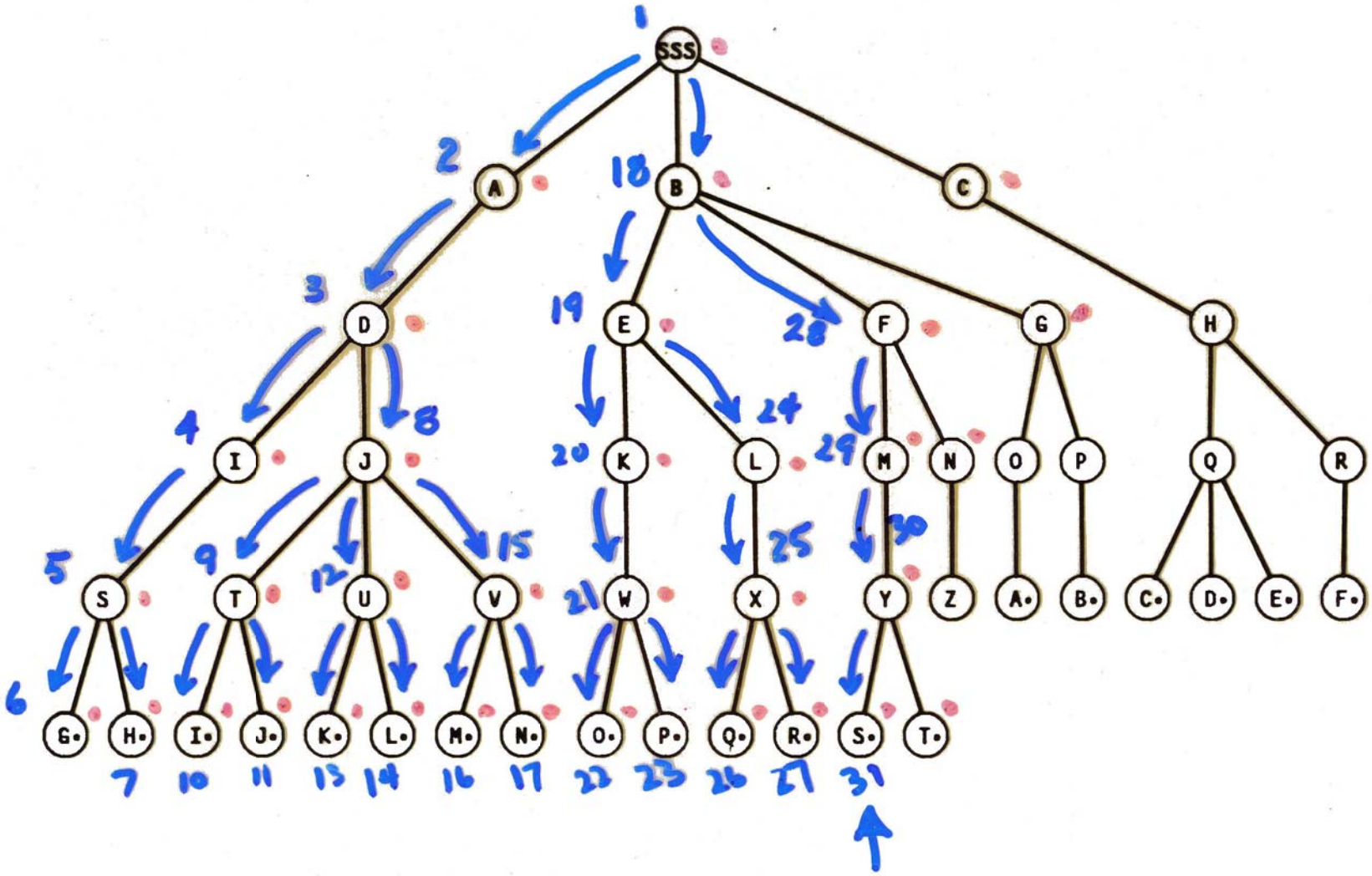
Depth-First Search: Append M and OPEN

Breadth-First Search: Append OPEN and M

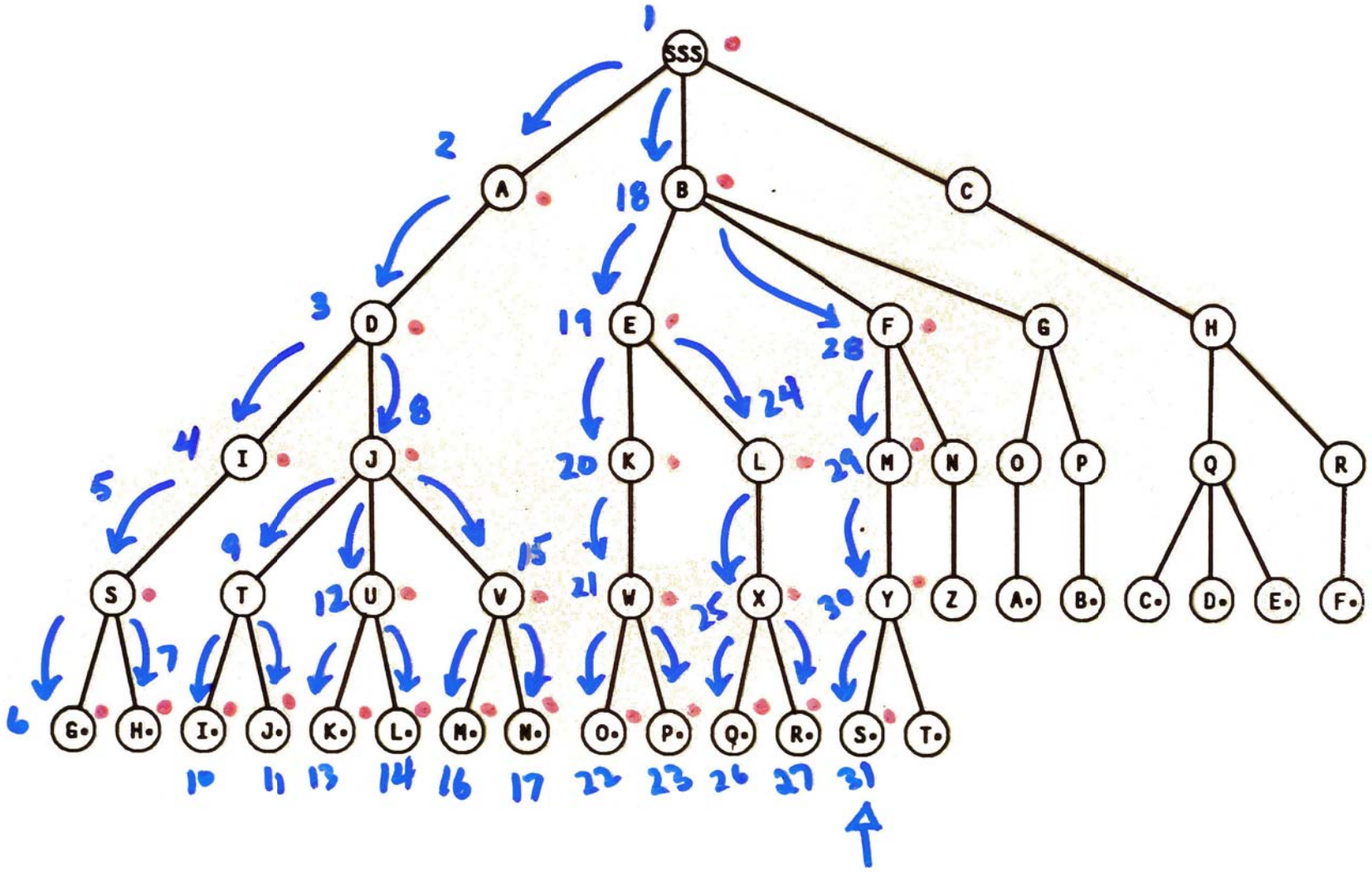
Search Example



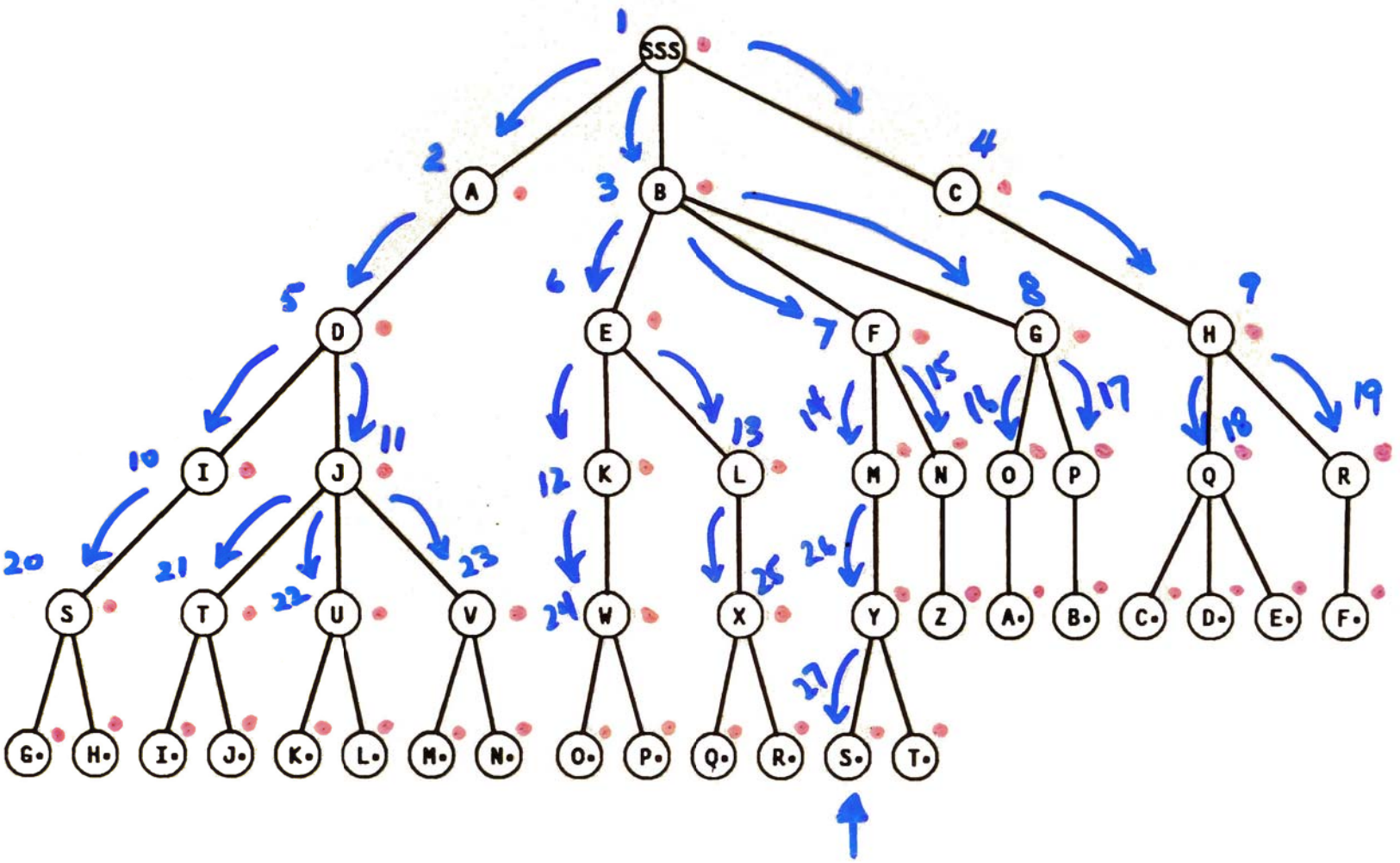
Search Example



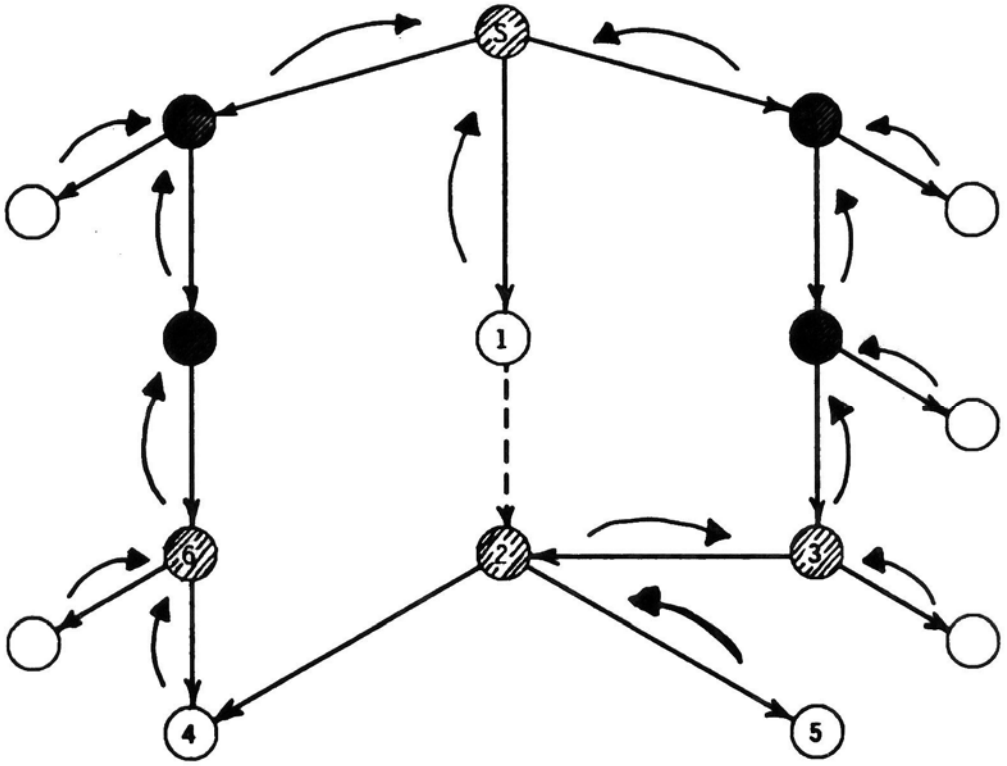
Search Example

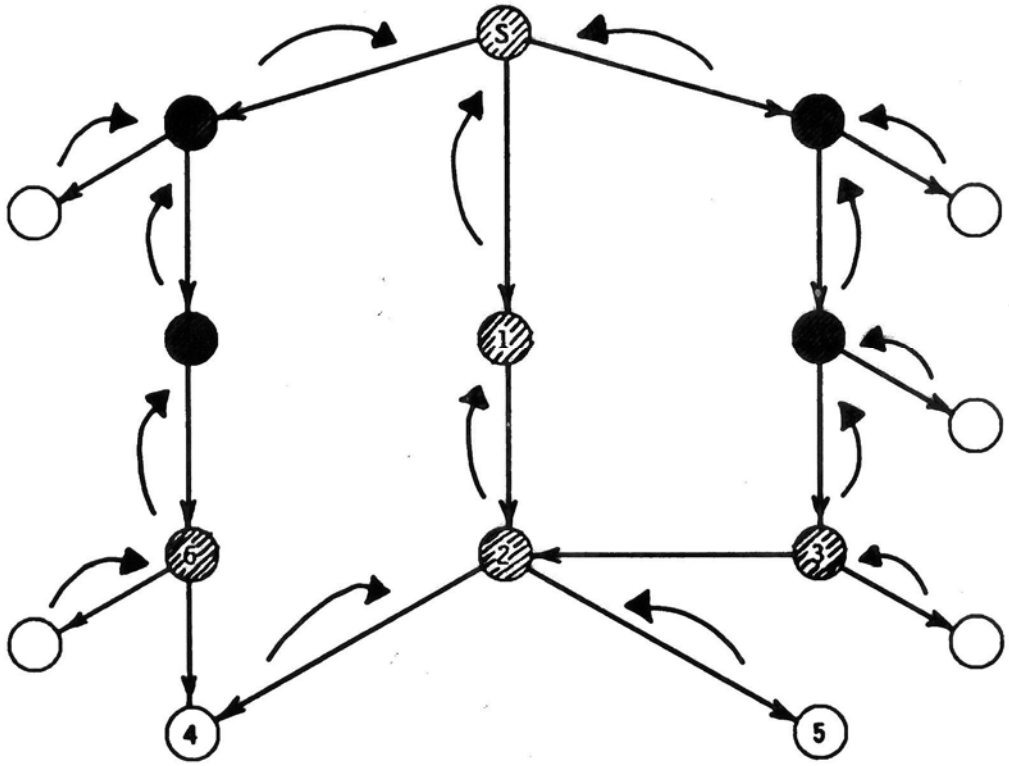


Search Example



Pointers





Heuristic Strategies

Use task-specific information to reduce search without (necessarily) sacrificing the guarantee of finding a minimal length path.

Usually try to minimize some combination of the cost of the path and the cost of the search required to find the path. (Averaged over all problems.)

Heuristic Power

Three ways to use heuristic information:

1) Deciding which node on OPEN to expand next.

2) Deciding which successors of a node to generate.

3) Deciding which nodes on OPEN to discard, or *prune*, from the search tree.

Evaluation Function (f) to estimate the "promise" of node n on OPEN. f(n) is the value.

$$f(n) = d(n) + w(n)$$

d(n) is the depth of node n in the search tree.

w(n) is an estimate of the distance of n from the goal.

Note: If f(n) = d(n), we get breadth-first search.

Algorithm A

$k(n_i,n_j)$ gives the *actual* cost of a minimal cost path between n_i and n_j. (k is undefined for nodes with no path between them.)

$k(n,t_i)$ is the cost of a minimal cost path from node n to some goal node t_i.

$h^*(n)$ is the minimum of all $k(n,t_i)$ over the set of goal nodes $\{t_i\}$. $h^*(n)$ is the cost of the minimal cost path from n to a goal node, so any path from n to a goal node that achieves $h^*(n)$ is an *optimal* path from n to a goal. (h^* is undefined for any node that has no accessible goal node.)

$g^*(s,n)$ is the cost of an optimal path from a given start node, s, to some arbitrary node n. (g^* is undefined for n not accessible from s.)

$f^*(n)$ is the cost of an optimal path from s to n plus the cost of an optimal path from n to a goal; that is, the cost of an optimal path from s constrained to go through n. ($f^*(s) = h^*(s)$ is the actual cost of an unconstrained path from s to a goal.)

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

We want a function f to be an estimate of f^*.

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

g is an estimate of g^* and h is an estimate of h^*.

g(n) could be the sum of the arc costs encountered while tracing the pointers from n to s. So, $g(n) \geq g^*(n)$.

h is called the *heuristic function* – this is the place where we can use task-specific information.

GRAPH-SEARCH using f as an evaluation function to order nodes on OPEN is called algorithm A.

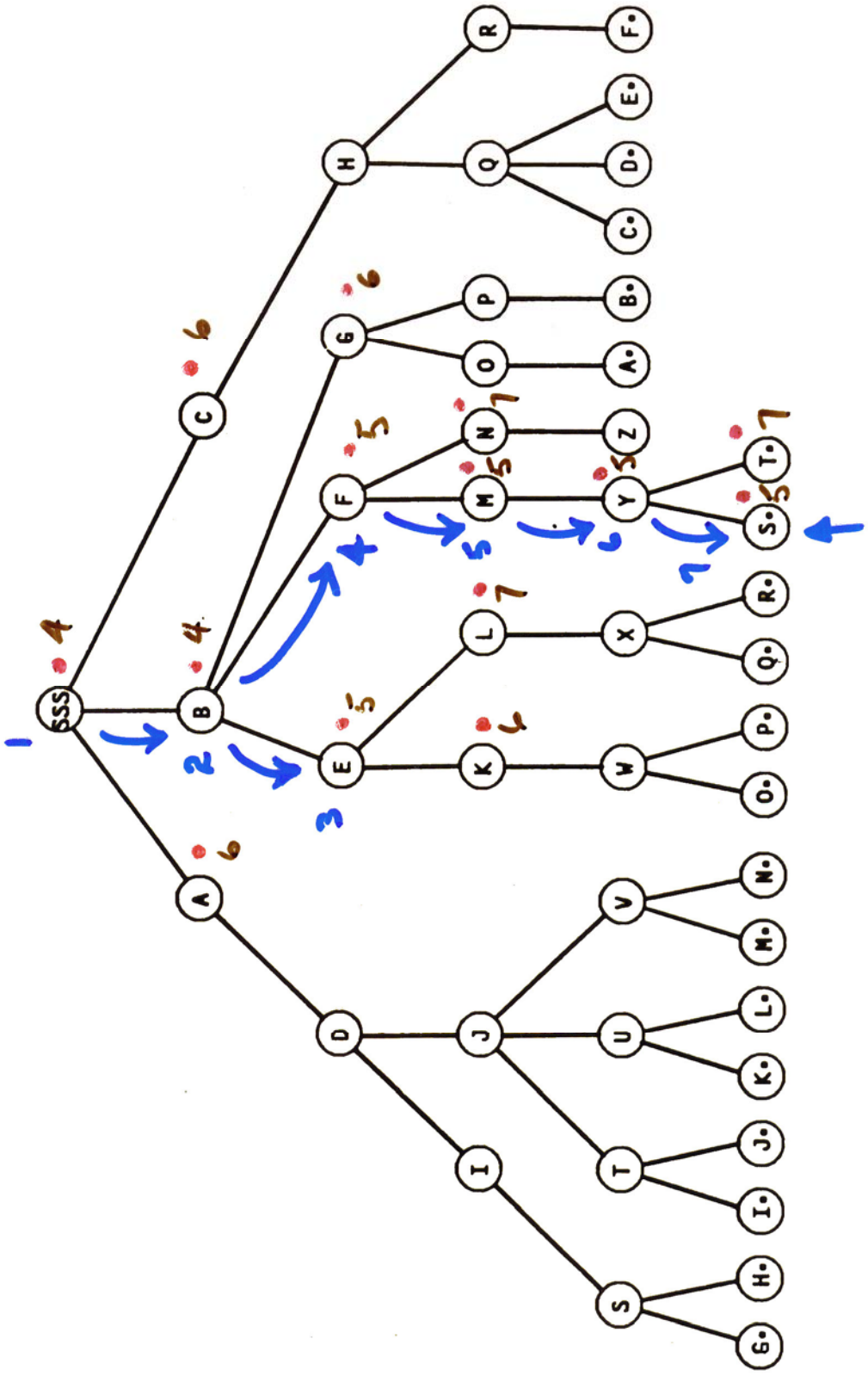
We can show that if algorithm A uses an h function such that h is a lower bound on h^* (i.e., $h(n) \leq h^*(n)$ for all nodes n), then it will find an optimal path to a goal. We call this version algorithm A^*.

Graph Search

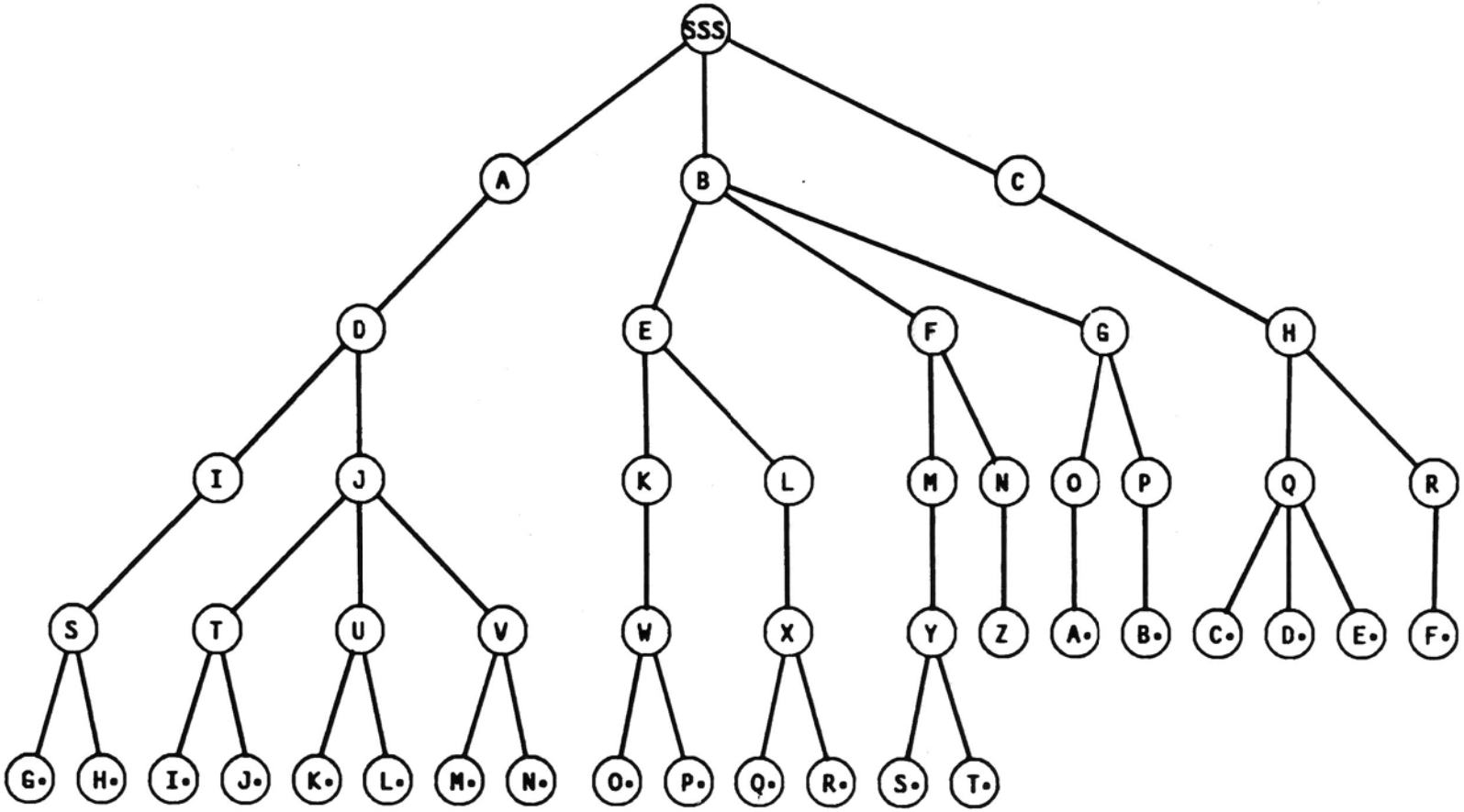
Path Cost / Depth-Bound / Partial Expansion

```
(GRAPH-SEARCH
  (LAMBDA (s)
    (PROG (OPEN CLOSED n M)
          (SET-COST s 0)
     **   (SET-DEPTH s 0)
          (SETQ OPEN (LIST s))
          (SETQ CLOSED NIL)
     LOOP (COND
            ((NULL OPEN)
              (RETURN 'FAIL)))
          (SETQ n (CAR OPEN))
          (COND
            ((GOAL-NODE n)
              (RETURN (CONS (GET-COST n)
                            (TRACE-PATH n s)))))
     **   (SETQ DEPTH (GET-DEPTH n))
     **   (COND
            ((EQ DEPTH DEPTH-BOUND)
              (SETQ OPEN (CDR OPEN))
              (SETQ CLOSED (CONS n CLOSED))
              (GO LOOP)))
          (SETQ M (EXPAND n ALLFLG))
          (COND
            ((OR ALLFLG (NULL M))
              (SETQ OPEN (CDR OPEN))
              (SETQ CLOSED (CONS n CLOSED))))
     **   (for m in M do
            (SET-DEPTH m (ADD1 DEPTH)))
          (SETQ M (POINTERS M n OPEN CLOSED))
          (SETQ OPEN (ORDER M OPEN))
          (GO LOOP))))
```

Search Example



Search Example



A^*: Best-First or Ordered Search

Admissibility

An algorithm is *admissible* if it always finds an optimal path (when a path exists).

Results:

1) GRAPH-SEARCH always terminates for finite graphs.

2) At any time before A^* terminates, there exists on OPEN a node n that is on an optimal path from s to a goal node, with $f(n) \leq f^*(s)$. (We assume $h(n) \geq 0$.)

3) If there is a path from s to a goal node, A* terminates. [Corollary: Any node n on OPEN with $f(n) < f^*(s)$ will eventually be selected for expansion by A*.]

4) A* is admissible. (If there is a path from s to a goal node, A* terminates by finding an optimal path.)

5) For any node n selected for expansion by A*, $f(n) \leq f^*(s)$

   Algorithm A_2^* is *more informed* than A_1^* if for all non-goal nodes, $h_2(n) > h_1(n)$.

6) If A_2^* is more informed than A_1^* then every node expanded by A_2^* is also expanded by A_1^* (at least).

A^*: Best-First or Ordered Search

Monotone Restriction: Similar to a triangle inequality

$$h(n_i) \leq h(n_j) + c(n_i,n_j)$$

n_j is a successor of n_i

7) If the monotone restriction is satisfied, then A^* has already found an optimal path to any node it selects for expansion; that is, if A^* selects n for expansion, and if the monotone restriction is satisfied, $g(n) = g^*(n)$.

8) If the monotone restriction is satisfied, the f values of the sequence of nodes expanded by A^* is nondecreasing.

Efficiency Boost: Keep track of F, the max f value of all nodes expanded. If a node n on OPEN has $f(n) < F$, then it will eventually be expanded. If several such nodes exist, try choosing the node from among them with the smallest g value – to try to enhance the chance that the first path discovered to a node will be an optimal path.

Relaxing the Optimality Requirement

$$f(n) = (1 - \omega)g(n) + \omega h(n)$$

Measures of Performance

X = Number of Nodes Expanded
P = Length of Solution Path
L = P/(Length of Minimal Length Path)
B = Effective Branching Factor: $B + B^2 + \ldots + B^p = X$

Bidirectional Search

Assume no heuristics available, average branching factor b, solution at depth d.

$$\text{Nodes} = \sum_{i}^{d} b^i \approx b^d \text{ for } b \geq 2$$

Say b = 4, d = 8, approx $7 \cdot 10^4$

Search in both directions to cut depth to d/2.

$$\text{Nodes} = 2\sum_{i}^{d/2} b^i \approx b^{d/2} \text{ for } b \geq 2$$

Say b = 4, d = 8, approx 512

Problems

Comparisons

At step d/2 *alone* require $b^{d/2} \cdot b^{d/2} = b^d$ comparisons.

Search Frontiers that don't meet in the middle.

Heuristic Power

1) Path Cost.

2) Number of nodes expanded to find the path.

3) Computational Effort required to compute h.

Tradeoffs

Search Effort vs Path Cost
(e.g., theorem proving)

Theoretically, the fact that performance is a function of relative error rather than of absolute error would seem to merit careful thought. What makes relative error more special? The "limits to growth" tabulated in the preceeding section are sobering: A* must be given a very accurate heuristic in order to guarantee good performance in the worst case. What about average case?

The symbolic result--a mapping from a lattice of heuristic "error" functions to a lattice of performance functions-- can serve as a definition for the (worst case) performance capability of the A* algorithm. The lattice representing heuristic functions exists independently of A*; it happens to be the domain of a particular function we have called XWORST. In words, the "knowledge itself" is distinct from the "knowledge engine itself", for "knowledge" in this special sense. Hence a similar XWORST function can perhaps be derived, assuming as the "knowledge engine" ordered depth-first search or the B* algorithm [Berliner 76] instead of A*. If you give an engine more knowledge, i.e., less errorful knowledge, then it performs better. But some engines can do more than others (or do it faster) with the knowledge they are given.

I greatfully acknowledge many fruitful discussions with Herbert Simon regarding this work.

References

1. Barstow, D., "A Knowledge-Based System for Automatic Program Construction", Proc. Intl. Joint Conf. Artificial Intelligence 77, Cambridge, Mass., August 1977.
2. Barstow, D., and E. Kant, "Observations on the Interaction of Coding and Efficiency Knowledge in the PSI Program Synthesis System", Proc. 2nd Intl. Conf. Software Engineering, San Francisco, October 1976, pp. 19-31.
3. Berliner, H., "The B* Tree Searching Algorithm: Best-First Search Combined with Branch and Bound", unpublished memo, Dept. of Computer Science, Carnegie-Mellon Univ., April 1976.
4. Doran, J., "New Developments of the Graph Traverser", in Machine Intelligence 2, E. Dale and D. Michie (eds.), American Elsevier Publ. Co., New York 1968.
5. Doran, J., and D. Michie, "Experiments with the Graph Traverser Program", Proc. Royal Society of London, Series A, Vol. 294, pp. 235-259, 1966.
6. Gaschnig, J., "The Worst Case Cost of A* as a Function on a Lattice of Heuristic Error Functions", Dept. of Computer Science Technical Report, Carnegie-Mellon Univ., July 1977. (1977a)
7. Gaschnig, J., Heuristic Search Performance and its Relation to Problem "Structure", Ph. D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ., forthcoming 1977. (1977b)
8. Green, C., Private communication. A best-first search mechanism for the program construction phase is being implemented. Other search reduction techniques will be used in conjunction with best-first search. April 1977.
9. Harris, L., "Heuristic Search Under Conditions of Error", Artificial Intelligence, Vol. 5, No. 3, pp. 217-234, North Holland Publishing Co., Amsterdam, 1974.
10. Hart, P., N. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Trans. Sys. Sci. Cybernetics, Vol. 4, No. 2, 1968.
11. Hayes, J., et. al., "A Quantitative Study of Problem-Solving Using Sliding Block Puzzles: the 'Eight Puzzle' and a Modified Version of the Alexander Passalong Test", Experimental Programming report no. 7, Experimental Programming Unit, University of Edinburgh 1965.
12. Hayes-Roth, F., and V. Lesser, "Focus of Attention in the Hearsay-II Speech Understanding System", Dept. of Computer Science Technical Report, Carnegie-Mellon Univ., January 1977.
13. Michie, D., "Strategy Building with the Graph Traverser", in Machine Intelligence 1, N. Collins and D. Michie (eds.), American Elsevier Publ. Co., New York 1967.
14. Michie, D., and R. Ross, "Experiments with the Adaptive Graph Traverser", in Machine Intelligence 5, B. Meltzer and D. Michie (eds.), American Elsevier, New York 1970.
15. Nilsson, N., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill Book Co., New York 1971.
16. Pohl, I., "First Results on the Effect of Error in Heuristic Search," Machine Intelligence 5, B. Meltzer and D. Michie (eds.), Edinburgh University Press, Edinburgh 1969.
17. Pohl, I., "Heuristic Search Viewed as Path-Finding in a Graph," Artificial Intelligence, Vol. 1, 1970.
18. Powers, G., et. al., "Optimal Strategies for the Chemical and Enzymatic Synthesis of Bihelical Deoxyrybonucleic Acids", J. American Chemical Soc., Vol. 97, p. 875, 1975.
19. Schofield, P., "Complete Solution of the Eight Puzzle", in Machine Intelligence 1, N. Collins and D. Michie (eds.), American Elsevier Publ. Co., New York 1967.
20. Vanderbrug, G., "Problem Representations and Formal Properties of Heuristic Search", Information Sciences, vol. 11, no. 4, 1976.
21. Woods, W., "Shortfall Scoring Strategies for Speech Understanding Control", in Speech Understanding Systems: Quarterly Technical Progress Report No. 6, Bolt Beranek and Newman report No. 3303, 1976.

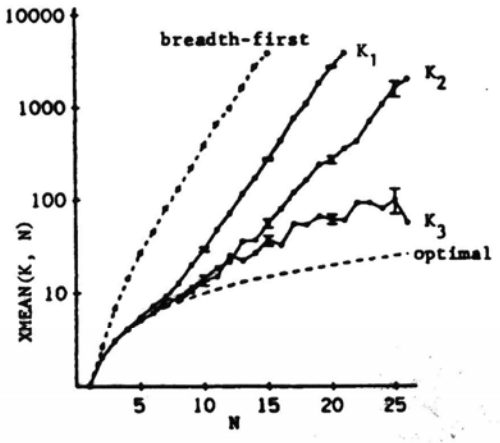


Figure 1. Number of nodes expanded vs. depth of goal for three 8-puzzle heuristics For optimal search, XMEAN(N) = N

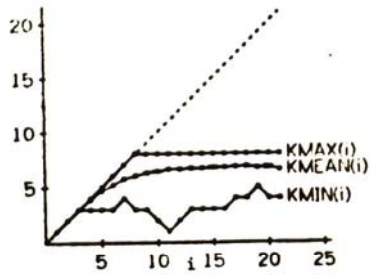


Figure 2. Heuristic estimate of distance to goal vs. actual distance, for heuristic K_1

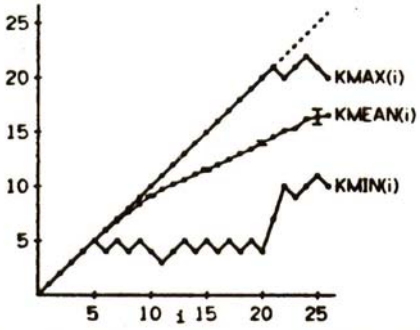


Figure 3. Same as figure 2, for heuristic K_2

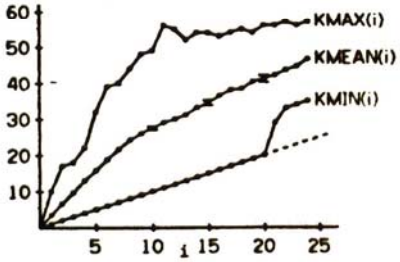


Figure 4. Same as figure 2, for heuristic K_3

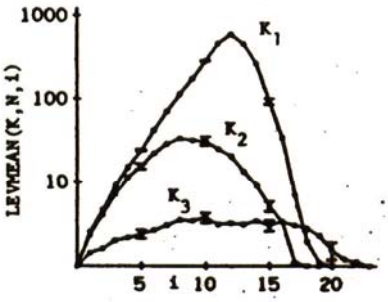


Figure 5. Number of nodes at level i in search tree; N = depth of goal = 20

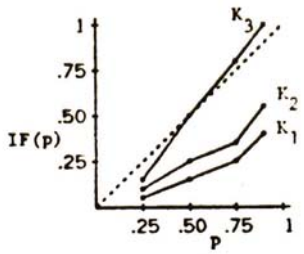


Figure 6. Interval fraction function for data in figure 5

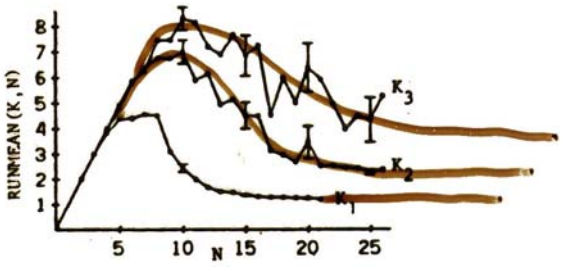


Figure 7. Mean run length vs. depth of goal

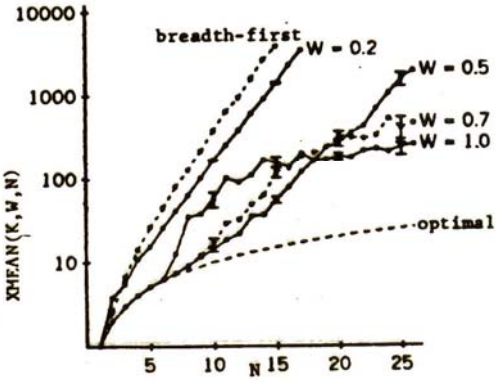


Figure 8. Number of nodes expanded vs. depth of goal; K_2 with different weight values

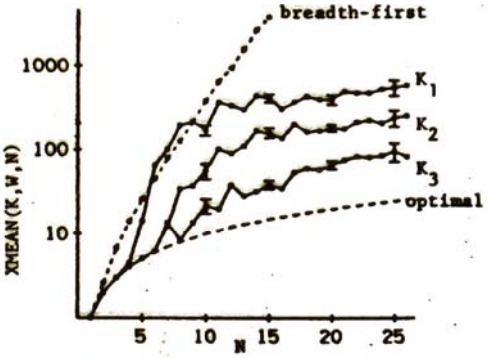


Figure 9. Number of nodes expanded vs. depth of goal, for 3 heuristics and W = 1.0 Compare with Figure 1.

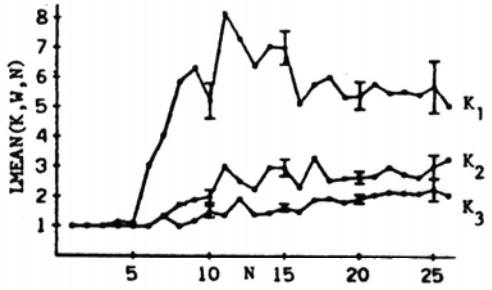


Figure 10. Solution path length ratio vs. depth of goal, at W = 1.0

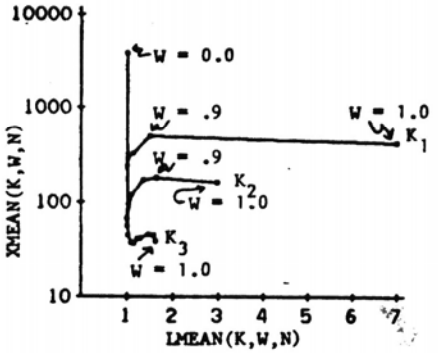


Figure 11. Cost vs. quality trade-off for fixed N = 15 as W varies

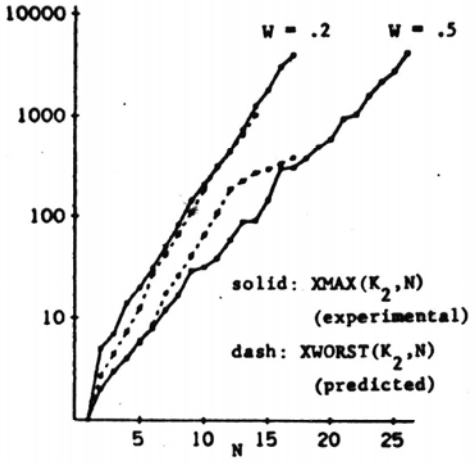


Figure 12. Predicted and measured values of worst case cost for heuristic K_2

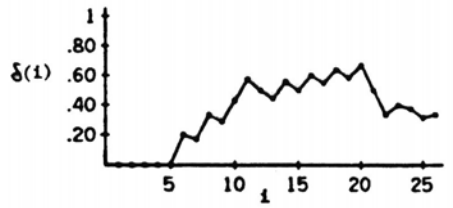


Figure 13. Relative error function for K_2 (from data in figure 3)

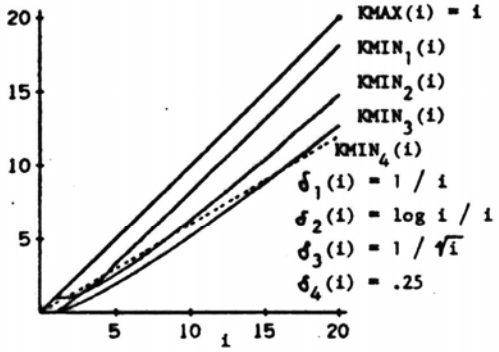


Figure 14. Fixed KMAX(i) and different KMIN(i) The KMIN functions have different assymptotic growth rates.

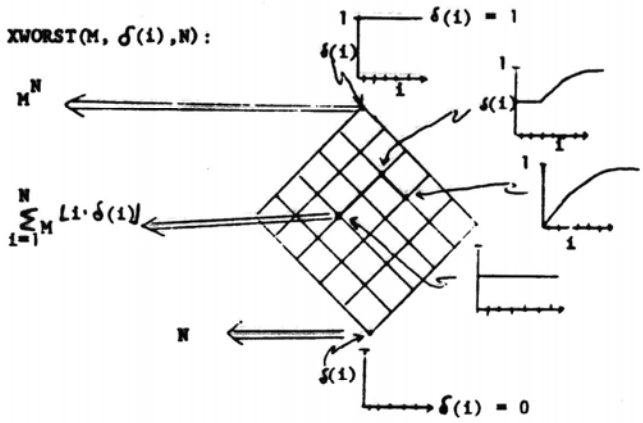
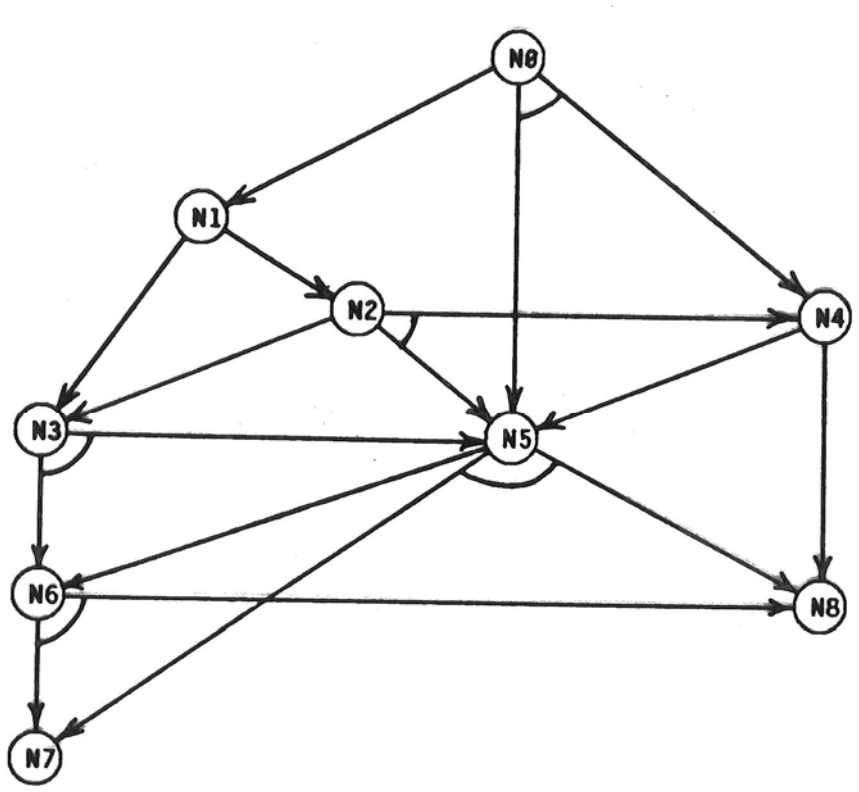


Figure 15. Finite depiction of infinite continuous lattice of heuristic functions and worst case cost as a function on the lattice. δ(KMIN,KMAX,i) is the relative error in a heuristic with bounding functions KMIN(i) and KMAX(i).

AND/OR Graph Search

Hypergraphs: *k-connectors* (hyperarcs) directed from one parent node to a set of successor nodes.



Solution Graph: G from node n to set of nodes N

Connector Costs

1) *Sum Costs:* $k(n,N) = c_n + k(n_1,N) + \ldots + k(n_i,N)$
2) *Max Costs:* $k(n,N) = c_n + \underset{i}{MAX}\left[k(n_1,N), \ldots k(n_i,N)\right]$

AND/OR Graph Search

```
(AO*
  (LAMBDA (s)
     (PROG (G G' n S m OLDCOST NEWCOST)
           (SETQ G (LIST s))
           (SET-COST s (H s))
           (COND
             ((TERMINAL-NODE s)
               (MARK s 'SOLVED)))
     OLOOP (SETQ G' (TRACE-MARKED-CONNECTORS s G))
           (COND
             ((SOLVED s)
               (RETURN (CONS (GET-COST s) G'))))
           (SETQ n (NSELECT G'))
           (SETQ G (APPEND G
                           (INSTALL (EXPAND n) n G)))
           (SETQ S (LIST n))
     ILOOP (COND
             ((NULL S) (GO OLOOP)))
           (SETQ m (MSELECT S G))
           (SETQ S (REMOVE m S))
           (SETQ OLDCOST (GET-COST m))
           (SETQ NEWCOST (REVISE-COST m))
           (COND
             ((OR (SOLVED m)
                  (NOT (EQUAL OLDCOST NEWCOST)))
               (SETQ S (APPEND S
                               (MARKED-PARENTS m)))))
           (GO ILOOP))))
```

AND/OR Graph Search

PSG: N0 Cost: 0, 1 Connector: N1, 2 Connector: N5 N4

Selected Nonterminal Leaf Node: N0
  Revising Cost of node: N0 0 → 3

PSG: N0 Cost: 3, 1 Connector: N1 [Marked], 2 Connector: N5 N4
        N1 Cost: 2, 1 Connector: N3, 1 Connector: N2

Selected Nonterminal Leaf Node: N1
  Revising Cost of node: N1 2 → 5, Adding Parents: N0
  Revising Cost of node: N0 2 → 4

PSG: N0 Cost: 4, 1 Connector: N1, 2 Connector: N5 N4 [Marked]
        N5 Cost: 1, 1 Connector: N6, 2 Connector: N7 N8
        N4 Cost: 1, 1 Connector: N5, 1 Connector: N8

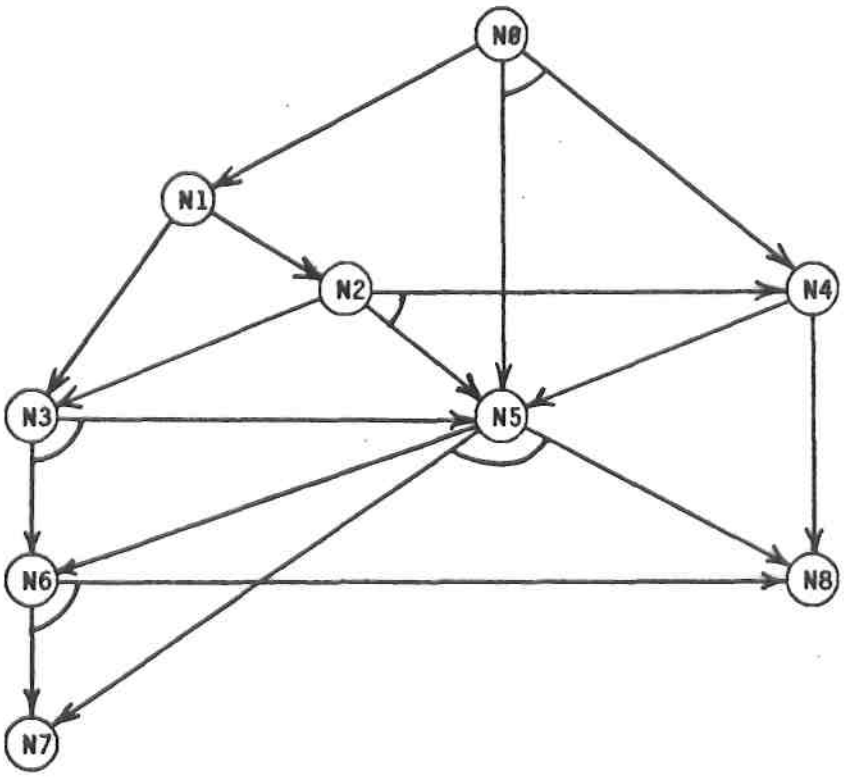
Selected Nonterminal Leaf Node: N5
  Revising Cost of node: N5 1 → 2 [Solved], Adding Parents: N0
  Revising Cost of node: N0 4 → 5

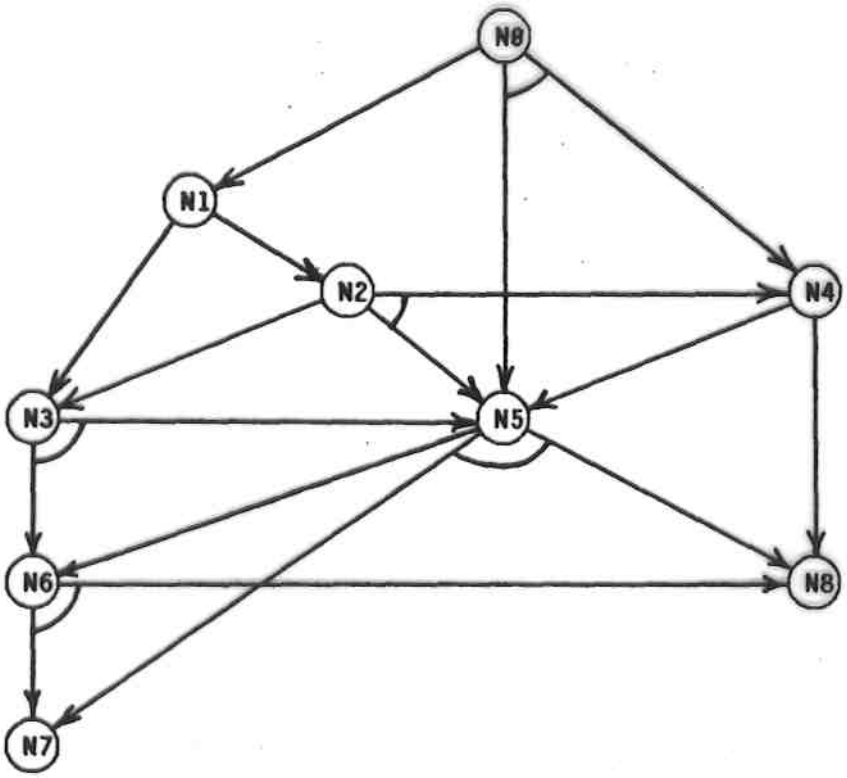
PSG: N0 Cost: 5, 1 Connector: N1, 2 Connector: N5 N4 [Marked]
        N5 Cost: 2, 1 Connector: N6, 2 Connector: N7 N8 [Marked]
        N4 Cost: 1, 1 Connector: N5, 1 Connector: N8
        N7 Cost: 0
        N8 Cost: 0

Selected Nonterminal Leaf Node: N4
  Revising Cost of node: N4 1 → [Solved], Adding Parents: N0
  Revising Cost of node: N0 5 → 5 [Solved]

PSG: N0 Cost: 5, 1 Connector: N1, 2 Connector: N5 N4 [Marked]
        N5 Cost: 2, 1 Connector: N6, 2 Connector: N7 N8 [Marked]
        N4 Cost: 1, 1 Connector: N5, 1 Connector: N8 [Marked]
        N7 Cost: 0
        N8 Cost: 0

N0: Property List
(CONNECTORS ((1 (NT) 1 NIL) (2 (N5 N4) 2 T))
          H 0 [Solved] T COST 5 PARENT NIL EXPANDED T)


N0
N1
N2
N3
N4
N5
N6
N7
N8


N0
N1
N2
N3
N4
N5
N6
N7
N8

AM

Lenat, 1976:

Open-ended heuristic search to find interesting concepts and conjecture relationships between them in elementary mathematics.

AM as a production system.

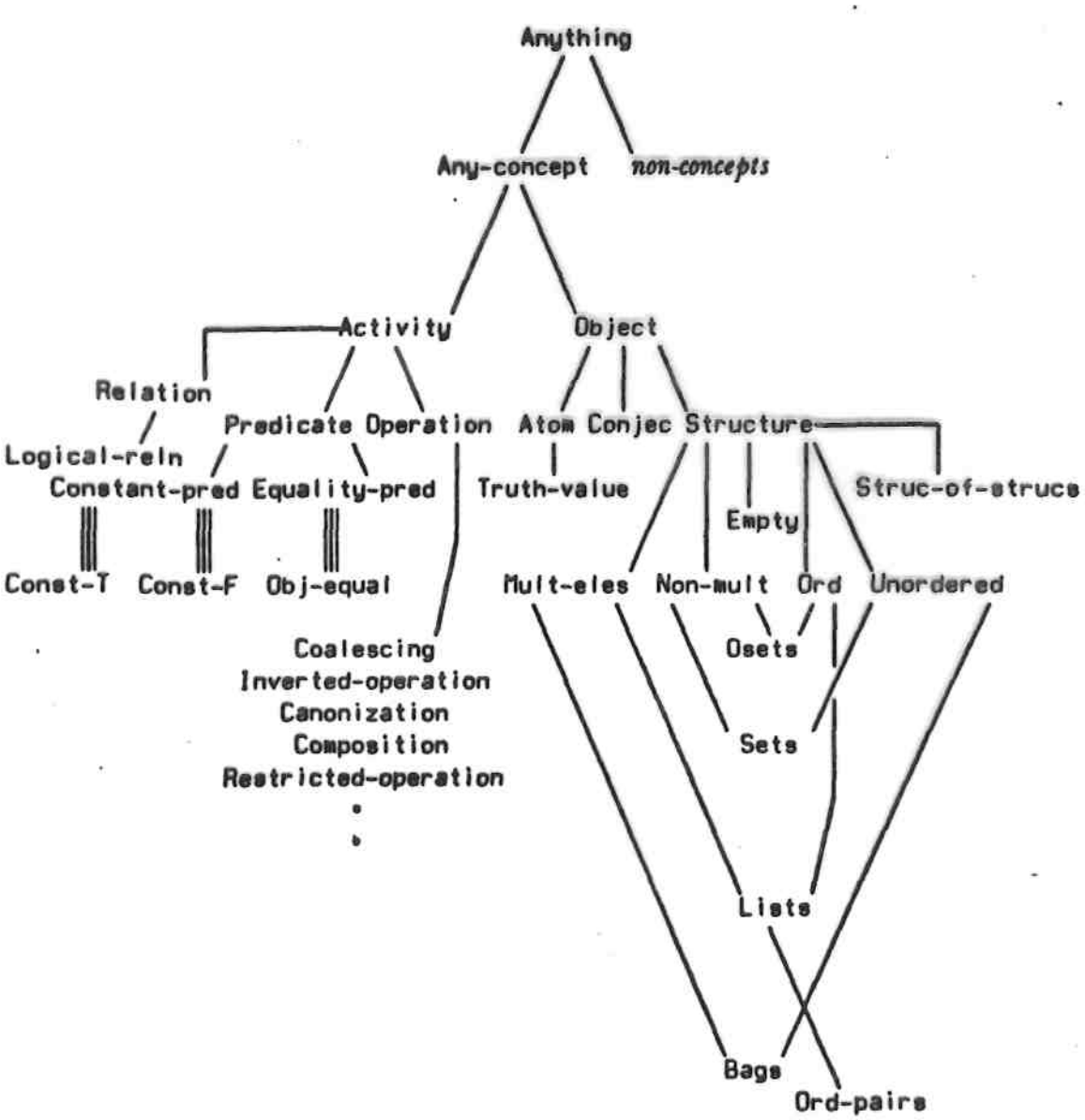
Database: Collection of *concepts*, represented as frames. Initially over 100 concepts given. Arranged in a generalization network. Approximately 25 slots per concept.

Examples:

Set, Ordered Set, List
Difference, Intersection
Equality

Rules: Heuristics for suggesting *(i)* new tasks to pursue, *(ii)* new concepts to create, or *(iii)* details to be filled in for an existing concept (and making new conjectures about concepts).

Control Strategy: *Agenda.* An ordered list of suggested tasks to execute (ordered by heuristic weights). This is essentially graph search controlled by heuristic ranking of utility of tasks. The agenda corresponds to the OPEN list. Repeat – Choose highest priority task on agenda and "execute" it. Also use priority rating to determine amount of resources to spend (350 $\rightarrow$ 35 CPU secs, 350 list-cells)



The diagram above represents the "topmost" concepts which AM had initially, shown connected via Specialization links (\) and Examples links (|||). The only concepts not diagrammed are *examples* of the concept Operation. There are 47 such operations.

Also, we should note that many entities exist in the system which are not themselves concepts. For example, the number "3", though it be an *example* of many concepts, is not itself a concept All entities which are concepts are present on the list called CONCEPTS, and they all have property lists (with facet names as the properties). In hindsight, this somewhat arbitrary scheme is regrettable. A more aesthetic designer might have come up with a more uniform system of representation than AM's.

AM

Sample Concept:

Name(s): Set, Class, Collection
Definitions:
Recursive: λ (S)
[S= {} or Set.Definition (Remove(Any-member(S),S))]
Recursive quick: λ (S) [S={} or Set.Definition (CDR(S))]
Quick: λ (S) [Match S with {…}]
Specializations: Empty-set, Nonempty-set, Singleton
Generalizations: Unordered-Structure, Collection,
Structure-with-no-multiple-elements-allowed
Examples:
Typical: {{}}, {A}, {A,B}, {3}
Barely: {}, {A, B, {C, {{{A, C, (3,3,9), <4,{B},A>}}}}}
Not-quite: {A,A}, (), {B,A}
Foible: <4,1,A,1>
Conjectures: All unordered-structures are sets.
Intuitions: **Geometric:** Venn Diagram.
Analogies: {set, set operations} ≡ {list, list operations}
Worth: 600 [on a scale of 0 – 1000]
View:
Predicate: λ (P) {x∈Domain(P) | P(x)}
Structure: λ (S)
Enclose-in-braces(Sort(Remove-multiple-elements(S)))
Suggest: If P is an interesting predicate over X,
Then consider {x∈X | P(x)}.
In-domain-of: Union, Intersection, Set-difference, Subset,
Member, Cartesian-product, Set-equality
In-range-of: Union, Intersection, Set-difference, Satisfying

AM: Sample Rules

If the current task was (Fill-in examples of X),
  and X is a predicate
  and more than 100 items are known in the domain of X,
  and at least 1 0 cpu secs were spent trying to randomly
    instantiate X,
  and the ratio of successes/failures is both >0 and less
    than .05
Then add the following task to the agenda:
  (Fill-in generalizations of X),
    for the following reason:
      “X is rarely satisfied; a slightly less restrictive
       concept might be more interesting.”
    This reasons's rating is computed as three times the ratio
      of nonexamples/examples.

AM: Sample Rules

If the current task was (Fill-in examples of F),
and F is an operation from domain space A into range B,
and more than 100 items are known examples of A
(in the domain of F),
and more than 10 range items (in B) were found by applying F
to these domain items,
and at least 1 of these range items is a distinguished member
(esp: extremum) of B,
Then (for each distinguished member 'b'$\in$B) create the following
new concept:

Name: F-inverse-of-b
Definition: λ (x) (F(x) is b)
Generalization: A
Worth: Average(Worth(A), Worth(F), Worth(B),
Worth(b), ||Examples(B)||)
Interest: Any conjecture involving both this concept and
either F or Inverse(F)

The reason for doing this is: "Worthwhile investigating those
A's which have an unusual F-value, namely, those whose
F-value is b"
The total amount of time to spend right now on all these new
concepts is computed as: Half the remaining cpu time in the
current task's time quantum.
The total amount of space to spend right now on all these new
concepts is computed as: The remaining space time quantum
for the current task.

AM: Sample Rules

If the current task is to fill in examples of the activity F,
One way to get them is to run F on randomly chosen examples
of the domain of F.

If the current task is to Check Examples of the operation F,
and F is an operation from domain space A into range B,
and F has at least 10 examples,
and a typical one of these examples is "<x $\rightarrow$ y>"
(so x$\in$A and y$\in$B)
and (Forsome Specialization BB of B), y is a BB,
and <u>all</u> examples of F (ignoring boundary cases) turn out
to be BB's,
Then print the following conjecture: "F(a) is always a BB, not
simply a B",
and add it to the Examples facet of Conjectures concept,
and add "<A $\rightarrow$ BB>" as a new entry to the Domain/Range
facet of F, replacing "<A $\rightarrow$ B>",
and if the user asks, inform him that the evidence for this
was that all ||Examples(F)|| examples of F (ignoring boundary
examples) turned out to be BB's,
and check the truth of this conjecture by running IF on boundary
examples of A.

AM: Sample Task

Activity: Fill in some entries
Facet: for the GENERALIZATIONS facet
Concept: of the PRIMES concept
Reasons: because

(1) There is only 1 known generalization of Primes, so far
(2) The worth rating of Primes is now very high
(3) Focus of attention: AM just worked on Primes
(4) Very few numbers are Primes; a slightly more plentiful concept might be more interesting.

Priority: 350 [on a scale of 0-1000]

** Task 65: **
Filling in examples of the following concept: "Divisors-of".

3 Reasons: (1) No known examples for Divisors-of so far.
(2) TIMES, which is related to Divisors-of, is now very interesting.
(3) Focus of attention: AM recently defined Divisors-of.

26 examples found, in 9.2 seconds. e.g.: Divisors-of(6)={1 2 3 6}.

** Task 66: **
Considering numbers which have very small sets of Divisors-of (i.e., very few divisors).

2 Reasons: (1) Worthwhile to look for extreme cases.
(2) Focus of attention: AM recently worked on Divisors-of.

Filling in examples of numbers with 0 divisors.
0 examples found, in 4.0 seconds.
Conjecture: no numbers have 0 divisors.

Filling in examples of numbers with 1 divisors.
1 examples found, in 4.0 seconds. e.g.: Divisors-of(1)={1}.
Conjecture: 1 is the only number with 1 divisor.

Filling in examples of numbers with 2 divisors.
24 examples found, in 4.0 seconds. e.g.: Divisors-of(13)={1 13}.
No obvious conjecture. This kind of number merits more study.
Creating a new concept: "Numbers-with-2-divisors".

AM had previously derived the notion of singleton, doubleton, etc. Above, AM was actually defining, e.g., the set of numbers which gets mapped into a doubleton of numbers, by the operation Divisors-of. I took the liberty of translating this into the above form, which uses, e.g., the number "2".

Filling in examples of numbers with 3 divisors.
11 examples found, in 4.0 seconds. e.g.: Divisors-of(49)={1 7 49}.
All numbers with 3 divisors are also Squares. This kind of number merits more study.
Creating a new concept: "Numbers-with-3-divisors".

How did AM notice that conjecture? It took a random example of Numbers-with-3-divisiors, say 49. Then it asked what other known concepts "49" was an example of. The two answers were: Odd-numbers and Perfect-squares. AM then tested these conjectures on the other ten examples just found. The only surviving conjecture was that all numbers-with-3-divisors are also perfect-squares.

** Task 67: **
Considering the square-roots of Numbers-with-3-divisors.

2 Reasons: (1) Numbers-with-3-divisors are unexpectedly also perfect Squares.
(2) Focus of attention: AM recently worked on Numbers-with-3-divisors.

All square-roots of Numbers-with-3-divisors seem to be Numbers-with-2-divisors.
e.g., Divisors-of(Square-root(169)) = Divisors-of(13) = {1 13}.

Formulating the converse to this statement. Empirically, it seems to be true.
The square of each Number-with-2-divisors seems to be a Number-with-3-divisors.
This is very unusual. It is not plausibly a coincidence. (Chance of coincidence is < .001)

Boosting interestingness factor of the concepts involved:
Interestingness factor of "Divisors-of" raised from 300 to 400.
Interestingness factor of "Numbers-with-2-divisors" raised from 100 to 600.
Interestingness factor of "Numbers-with-3-divisors" raised from 200 to 700.

USER: Call the set of numbers with 2 divisors "Primes".

** Task 68: **
Considering the squares of Numbers-with-3-divisors.

2 Reasons: (1) Squares of Numbers-with-2-divisors were interesting.
(2) Focus of attention: AM recently worked on Numbers-with-3-divisors.

•
•
•

This gap in the sequencing — from task 67 to task 79 — eliminates some tangential and boring tasks. See page 19 for an explanation.

** Task 79: **
Examining TIMES^{-1}(x), looking for patterns involving its values.

2 Reasons: (1) TIMES^{-1} is related to the newly-interesting concept "Divisors-of".
(2) Many examples of TIMES^{-1} are known, to induce from.

Looking specifically at TIMES^{-1}(12), which is { (12) (2 6) (2 2 3) (3 4) }.
13 conjectures proposed, after 2.0 seconds.
e.g., "TIMES^{-1}(x) always contains a bag containing only even numbers".
Testing the conjectures on other examples of TIMES^{-1}.

5 false conjectures deal with even numbers.
AM will sometime consider the restriction of TIMES^{-1} to even numbers.

Only 2 out of the 13 conjectures are verified for all 26 known examples of TIMES^{-1}:

Conjecture 1: TIMES^{-1}(x) always contains a singleton bag.
e.g., TIMES^{-1}(12), which is { (12) (2 6) (2 2 3) (3 4) }, contains (12).
e.g., TIMES^{-1}(13), which is { (13) }, contains (13).

Creating a new concept, "Single-times".
Single-times is a relation from Numbers to Bags-of-numbers.
Single-times(x) is all bags in TIMES^{-1}(x) which are singletons.
e.g., Single-times(12)={ (12) }.
e.g., Single-times(13)={ (13) }.

Conjecture 2: TIMES^{-1}(x) always contains a bag containing only primes.
e.g., TIMES^{-1}(12), which is { (12) (2 6) (2 2 3) (3 4) }, contains (2 2 3).
e.g., TIMES^{-1}(13), which is { (13) }, contains (13).

Creating a new concept, "Prime-times".
Prime-times is a relation from Numbers to Bags-of-numbers.
Prime-times(x) is all bags in TIMES^{-1}(x) which contain only primes.
e.g., Prime-times(12)={ (2 3 3) }.
e.g., Prime-times(13)={ (13) }.

AM: Summary

Main Limitation: Unable to examine heuristics and devise new ones. Eventually its built-in heuristics were too general to be of much use.

Is **AM** …

Intelligent?

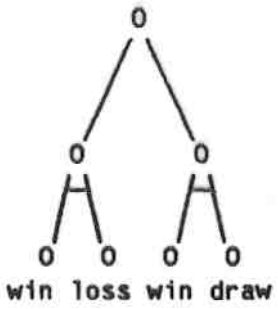
Creative?

Learning?

Useful?

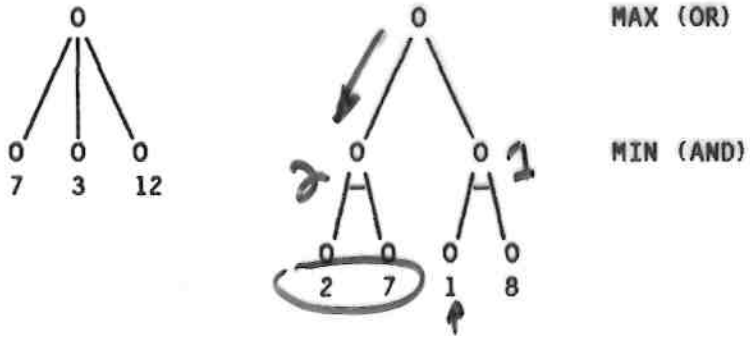
Searching Game Trees

Game Trees are similar to AND/OR Trees: You may select any single move when it is your turn (OR), but you must be able to respond to all possible replies by your opponent (AND).



Static Evaluation: It is not generally feasible to search forward to the end of a game to find "goal states." Why?
Hence search forward a few moves and evaluate the *likelihood* of a win from that point. How?
Then try to use the information found by "looking ahead" to find a good first move (which may well not turn out to be on the optimal path to a win).

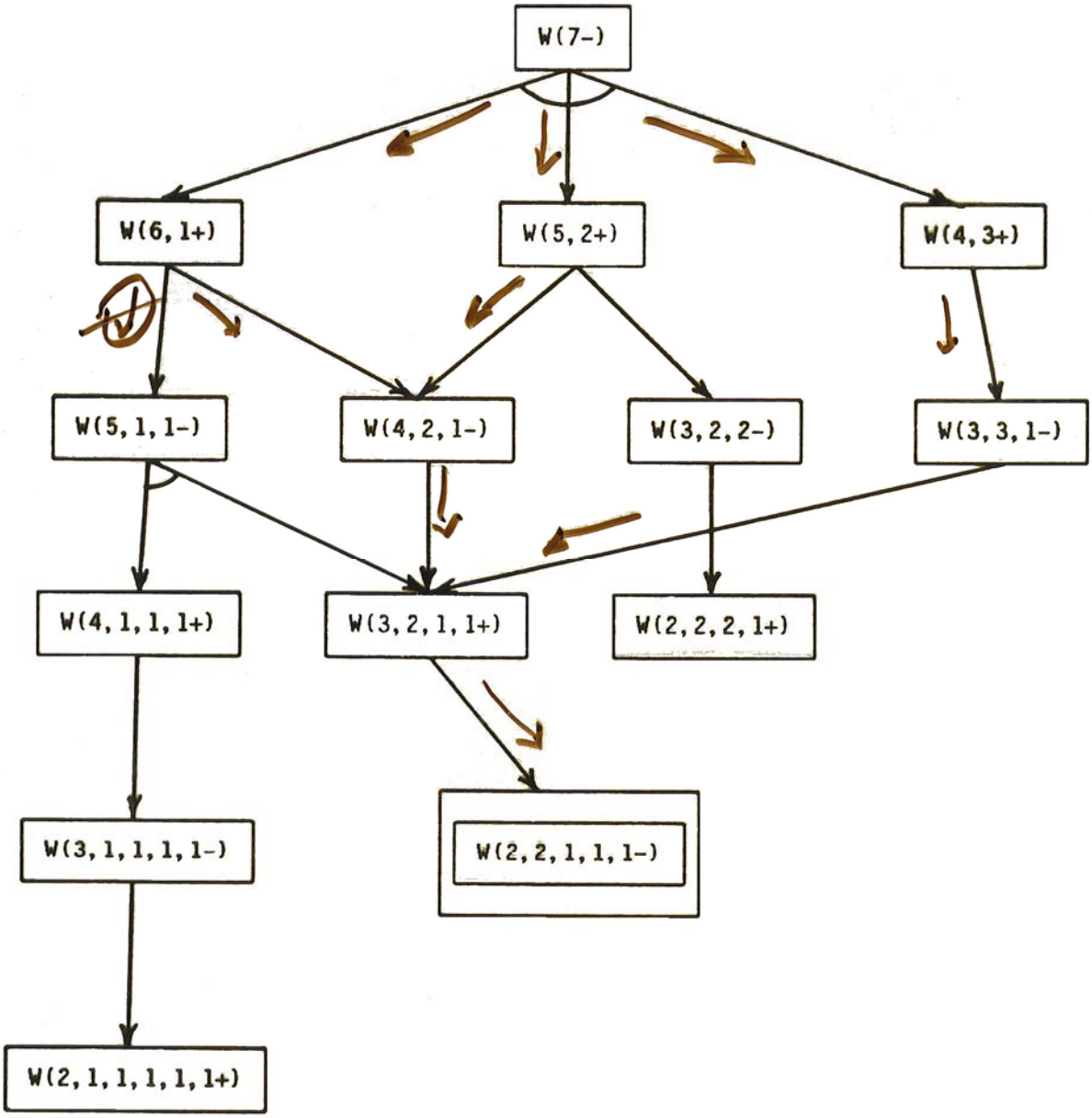
MINIMAX Procedure:



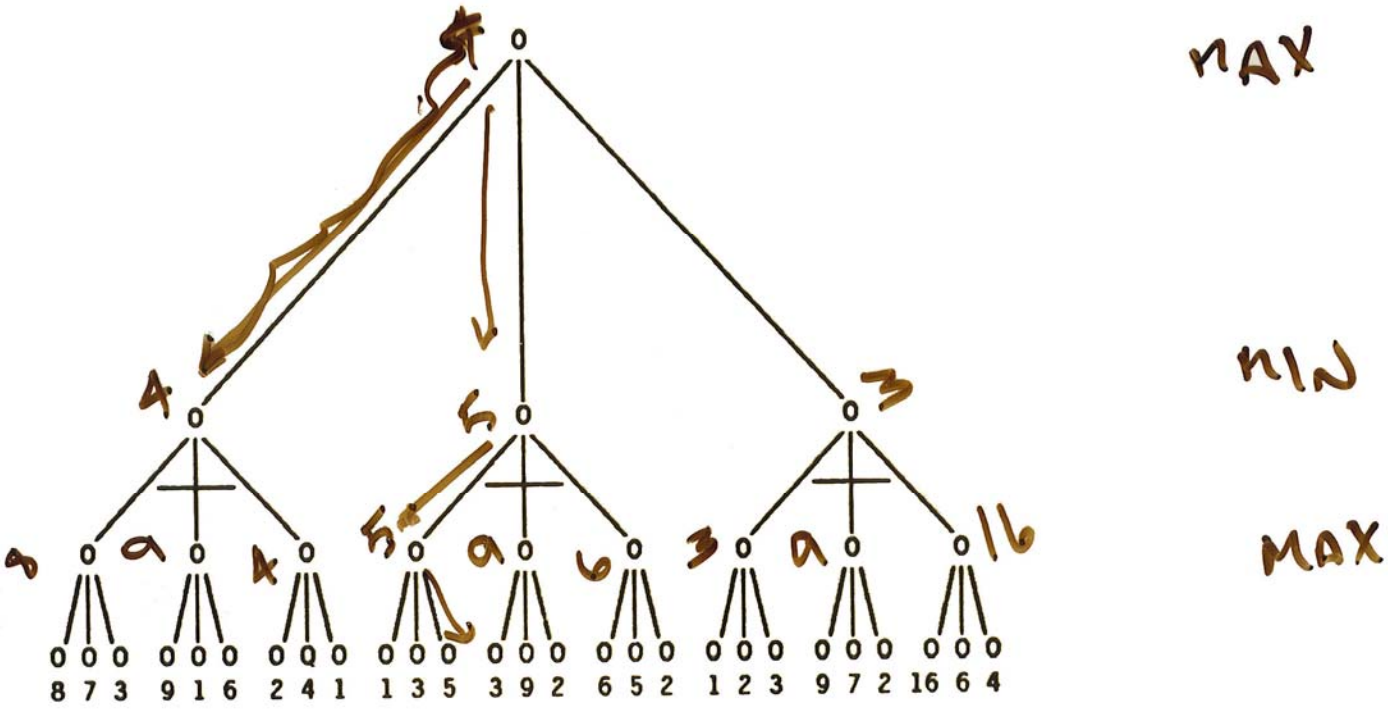
Backed-Up values

Searching Game Trees

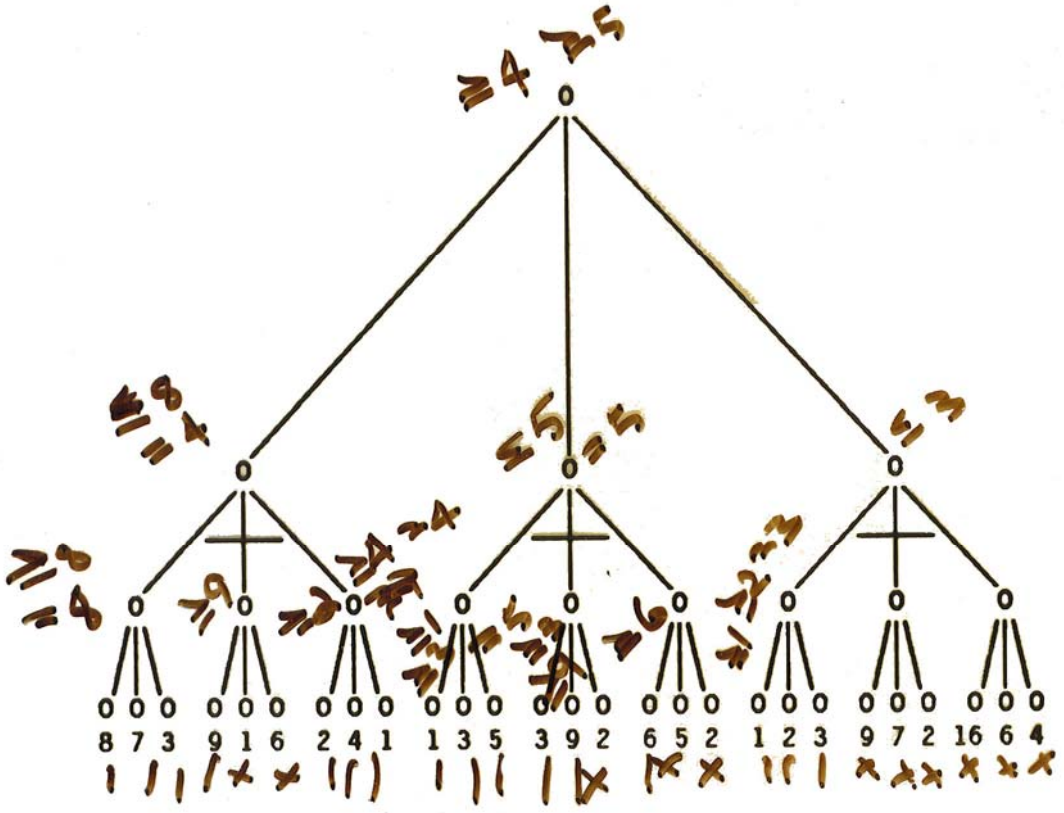
Grundy's Game: Divide a stack into two stacks that are unequal



Searching Game Trees



Searching Game Trees



Searching Game Trees

ALPHA-BETA Procedure: Generates same result as MINIMAX – but more efficiently.

Interleave tree generation and evaluation.

Compute upper and lower bounds for backed-up value at each node n such that, unless α < Value(n) < β, the game will never pass through node n.

α value of a MAX node is set to current largest backed-up value of its successors. (Note that α value of a node can *never* decrease.)

β value of a MIN node is the current smallest backed-up value of its successors. (Note that β value of a node can *never* increase.)

Prune whenever:

1. For a MIN node: $\beta \leq \alpha$ for any MAX node ancestor (α cutoff).
2. For a MAX node: $\alpha \geq \beta$ for any MIN node ancestor (β cutoff).

Effectiveness:

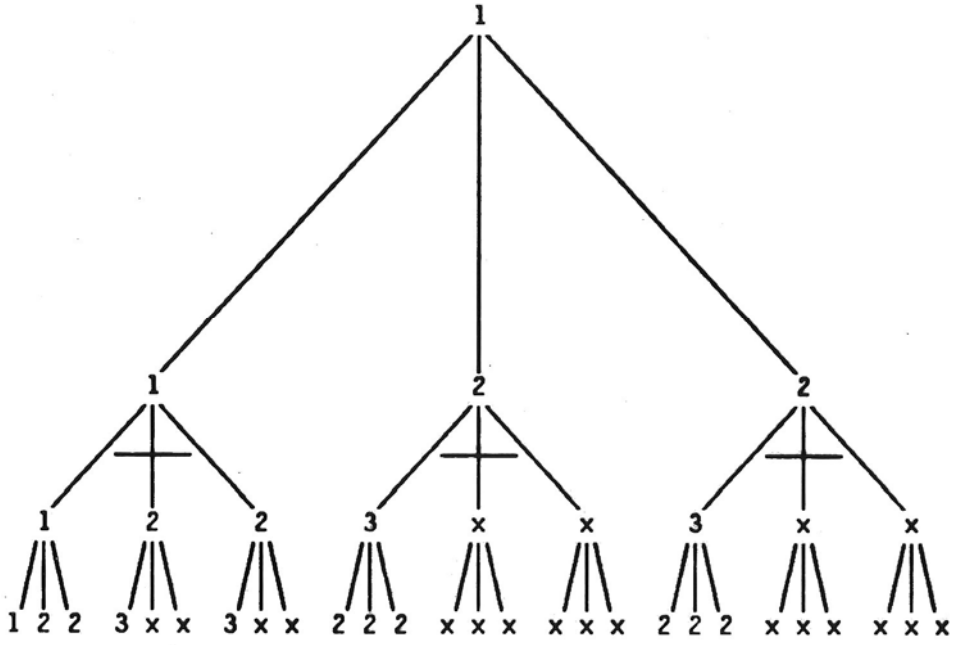
Assume a regular tree, depth d, branching factor b. MINIMAX would have to examine b^d nodes.

In the best case (where successors are generated in order of their true backed-up values), ALPHA-BETA examines approximately $2b^d/^2$ nodes.

In the worst case, ...
See Knuth and Moore (1 975) for details.

Searching Game Trees

ALPHA-BETA: Critical Nodes



Type 1: The root node and all first successors of type 1 nodes.

Type 2: All further successors (except the first) of type 1 nodes and all successors of type 3 nodes.

Type 3: First successors of type 2 nodes.

Searching Game Trees

Improving Effectiveness:

Move Ordering: preliminary evaluation estimate
Refutation Moves:
Backed-up Evaluation: using extra information
Satisficing: setting a “satisfactory” value to be obtained
Iterative Deepening:
Quiescence Extensions: the horizon effect
Limiting Breadth: (perhaps variable according to ranking)
Goal-directed Move Generation:

Searching Game Trees

```
(MINIMAX
  (LAMBDA (BOARD ALPHA BETA DEPTH)
    (PROG (BOARDS MOVE NEXT-BOARD
           NEW-VALUE MOVES RESULT)
          (COND
            ((OR (QUIET BOARD)
                 (NOT (SETQ BOARDS
                            (PLAUSIBLE-MOVES BOARD))))
              (RETURN
                (LIST (STATIC-VALUE BOARD DEPTH)
                      NIL))))
          (SETQ MOVE 1)
          (SETQ RESULT (LIST ALPHA NIL))
     LOOP (COND ((NULL BOARDS) (RETURN RESULT)))
          (SETQ NEXT-BOARD (CAR BOARDS))
          (SETQ NEW-VALUE (MINIMAX NEXT-BOARD
                                   (MINUS BETA)
                                   (MINUS ALPHA)
                                   (ADD1 DEPTH)))
          (SETQ MOVES (CADR NEW-VALUE))
          (SETQ NEW-VALUE (IMINUS (CAR NEW-VALUE)))
          (COND
            ((IGREATERP NEW-VALUE ALPHA)
              (SETQ ALPHA NEW-VALUE)
              (SETQ RESULT
                (LIST NEW-VALUE
                      (CONS MOVE MOVES)))))
          (COND
            ((IGEQ ALPHA BETA)
              (PRINT-MESSAGE BOARDS DEPTH)
              (RETURN RESULT)))
          (SETQ BOARDS (CDR BOARDS))
          (SETQ MOVE (ADD1 MOVE ))
          (GO LOOP))))
```

Searching Game Trees

MINIMAX Execution Trace: Nilsson, p. 124.

```
   Node evaluated, value 0
   Node evaluated, value 5
   Node evaluated, value -3
MINIMAX trims 1 at level 5
   Node evaluated, value 3
MINIMAX trims 2 at level 4
   Node evaluated, value 5
   Node evaluated, value 2
MINIMAX trims 1 at level 4
MINIMAX trims 1 at level 2
   Node evaluated, value 5
   Node evaluated, value 1
   Node evaluated, value -3
MINIMAX trims 1 at level 5
   Node evaluated, value -5
MINIMAX trims 1 at level 5
MINIMAX trims 1 at level 3
   Node evaluated, value 2
   Node evaluated, value 3
   Node evaluated, value -3
MINIMAX trims 1 at level 5
   Node evaluated, value -1
MINIMAX trims 1 at level 5
   Node evaluated, value 0
MINIMAX trims 1 at level 5
   Node evaluated, value 4
   Node evaluated, value 5
MINIMAX trims 1 at level 2
   Node evaluated, value -1
MINIMAX trims 1 at level 5
MINIMAX trims 1 at level 3

Backed-up value and Move Sequence: 1 (2 1 1 1 1 2)
```

GPS

General Problem Solver: The first problem-solving program to separate its general problem-solving methods from knowledge specific to the task at hand.

Describe a task as a triple <S,G,O>

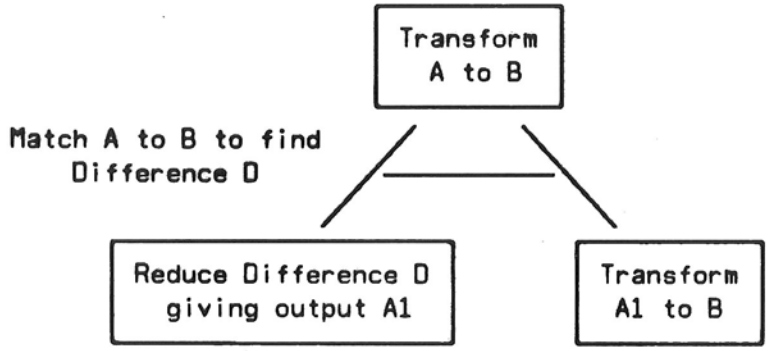
S = Initial Object (State)
G = Goal Object (State)
O = Operators (Rules)

Transform S into G using O.

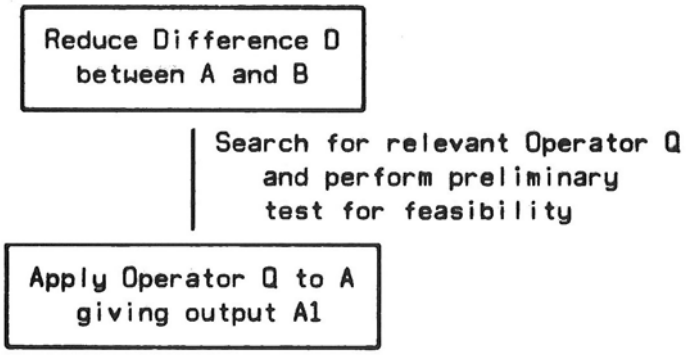
GPS uses a problem-reduction method called **Means-Ends Analysis.** The search is organized around the goal of reducing the *difference* between the initial object and the goal object.

GPS: Goal Types and Methods

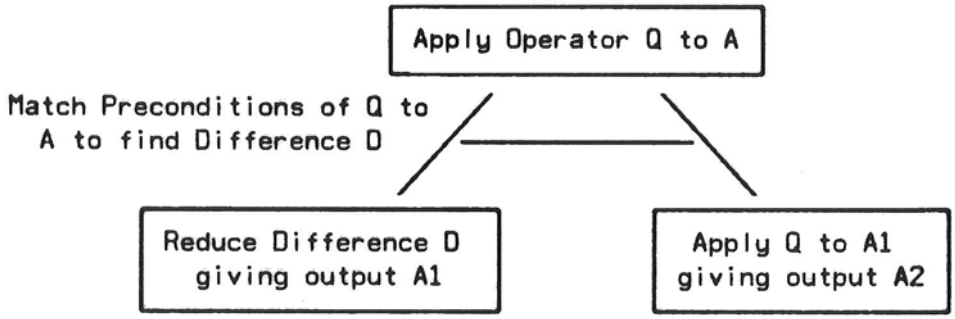
1. *Transform* object A into object B



2. *Reduce* difference D between object A and object B



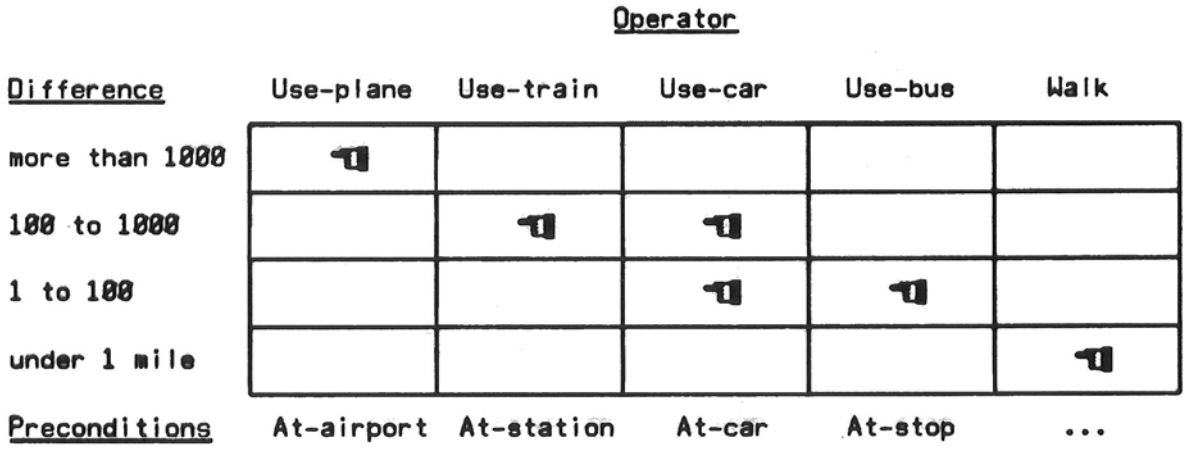
3. *Apply* operator Q to object A



GPS

Table of Connections: Order the differences by difficulty and associate a list of operators with each difference.

Implicit operator selection



Example:

Initial Object: At(Fred,Home-in-Halifax)
Final Object: At(Fred,Wreck-Cove-in-Vancouver)

d greater than 1000	Fly	At(Fred,Airport)
d between 1 and 100	Drive	At(Fred.Car)
d less than 1	Walk	
	(Walk to Car, Drive to Airport)	
d less than 1	Walk	
	(Walk to Airplane, Fly to Vancouver)	
d between 1 and 100	Drive	At(Fred,Car) ??
	Take Bus	At(Fred,Stop)
d less than 1	Walk	
	(Walk to Stop, Take Bus to UBC)	
d less than 1	Walk	
	(Walk to Wreck Cove)	

Predicate Calculus

Motivation: Need a language for representing facts about the world. State vectors are not always appropriate.

Example: How to represent...

All 375A students are going to get rich.
Alice is a 375A student.

Is Alice a 375A student?
Is Alice going to get rich?

Syntax:

Predicate Symbols: `MARRIED`
Function Symbols: `father`
Variable Symbols: `x y z`
Constant Symbols: `FRED ALICE TREE`

Term: Constant | Variable | Function($Term_1$, $Term_2$, ...)
Atomic Formula: Predicate-Symbol($Term_1$, $Term_2$, . . .)

Example:

```
MARRIED(KRIS,RITA)
MARRIED(father(KRIS),mother(KRIS))
```

Predicate Calculus

Semantics: To give an atomic formula (wff – well-formed formula) meaning, we must *interpret* it as making an assertion about a domain (e.g., set of integers, 8-puzzle configurations, set of mathematicians,...). Given a wff and an interpretation (or model), we can assign a value T or F to it.

Example: `P(A,f(B,C))` D = set of integers.

Assign some element of D to every constant symbol in the wff.

`A` → `integer 2`
`B` → `integer 4`
`C` → `integer 6`

Assign a function over D to every function symbol in the wff.

`f` → `addition function`

Assign a relation among the elements of D to every predicate symbol in the wff.

`P` → `greater-than relation`

Under this interpretation, the wff states “2 is greater than the sum of 4 plus 6.” The value of the wff is F. (Suppose we re-interpret the wff so that `A` → `11`...)

Predicate Calculus

Connectives: ¬ ∧ (&) ∨ → (⊃)

Examples

CLEARTOP(BLOCK1) ∧ ONTABLE(BLOCK1) ∧ HANDEMPTY

HOLDING(BLOCK1) ∨ HOLDING(BLOCK2)

¬HOLDING(BLOCK3)

A → B (¬A ∨ B) Implication: Antecedent → Consequent

A	B	0	1
0		1	1
1		0	1

Literal: Atomic Formula | ¬ Atomic Formula

Propositional Calculus: Can't say *"All terminals have keyboards."*

Quantifiers:

Universal: ∀ *All* Existential: ∃ *There exists*

Examples

All terminals have keyboards.
(∀x) [TERMINAL (x) → HAS-KEYBOARD(x)]

There is a terminal that does not have a keyboard.
(∃x) [TERMINAL(x) ∧ ¬HAS-KEYBOARD(x)]

For *every integer, there exists an integer that is larger.*
(∀x)(∃y) [LARGER(y,x)]

Interpretation Scope
Quantified Variable Bound Variables
Free Variables

Predicate Calculus

wff: Literals | (Quantifier Variable)[wff]

Sentence: wff with all variables bound

Ground Instance or Ground wff: No variables

First Order Predicate Calculus: Cannot quantify over predicate symbols or function symbols.

Cannot say: $(\forall x)(\exists P)[P(father(x)) \rightarrow P(x)]$

Example wff:

$$(\forall x)[MAILMAN(x) \rightarrow (\exists y)\{DOG(y) \wedge HAS\text{-}BITTEN(y,x)\}]$$

Example non-wffs:

$\neg f(A)$

$Q\{f(A),[p(B) \rightarrow Q(C)]\}$

$f[P(A)]$
$\wedge A(x)$

Overview

Given an *interpretation* over a finite domain, there are ways of establishing the truth value for any wff (**truth table method**).

Valid wff: Truth value is T for *all* interpretations **(Tautology** if a ground wff).

Equivalent wffs: Truth values are the same, regardless of their interpretation.

$\neg(A \wedge B) \equiv \neg A \vee \neg B$

A wff is **satisfiable** if there is some interpretation in which the wff is true. (The interpretation *satisfies* the wff.)

$(\forall x)P(x)$ $\quad$ $P(A) \wedge \neg P(A)$

Predicate Calculus

Can always determine the validity of a wff that contains no quantifiers.

Not always possible if quantifiers are involved (Checking procedure may not terminate). Hence PC is *undecidable.*

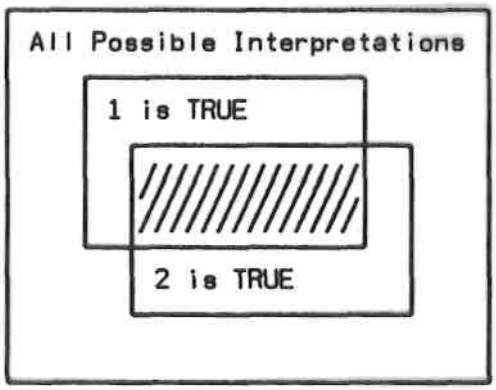
But...Can always prove the validity of valid wffs; Cannot always prove the invalidity of invalid wffs. Hence PC is *semidecidable.* Also, can prove the validity of certain classes of formulas with quantifiers *(decidable subclasses).*

A wff X **logically follows** from a set of wffs S provided that *every* interpretation that satisfies each wff in S also satisfies X.

$P(A)$ *logically follows from* $(\forall x)\ P(x)$
$(\forall x)P(x)$ *does not logically follow from* $P(A)$

1. $(\forall x)[P(x) \vee Q(x)]$
2. $\neg Q(A)$

For what interpretations does $P(A)$ logically follow from 1 and 2?



Predicate Calculus

Inference Rules: Applied to wffs to produce new wffs

Theorem: A wff derived from application of inference rules to a set of wffs. (The sequence of rule applications is a proof.)

Sound Rules: Any theorem derivable from a set also logically follows from the set.

Complete Set of Rules: All wffs that logically follow are theorems (i.e., can prove any provable theorem).

Natural Deduction: Elimination and Introduction [FOL]

Modus Ponens: Given X, $X \rightarrow Y$, infer Y

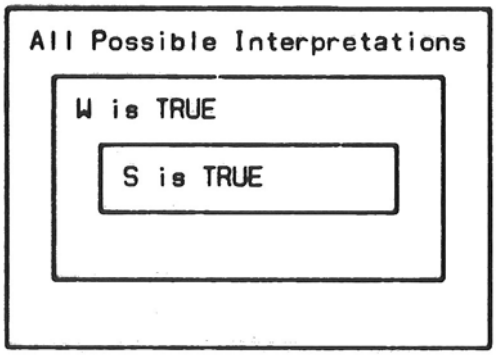
Universal Specialization: Given $(\forall x)P(x)$, infer $P(A)$

Resolution: (Robinson, 1965) Complete.

Resolution

Overview

Problem: To show that wff W logically follows from set of wffs S.
Method: Show that the set S $\cup$ {$\neg$W} is unsatisfiable.



This is called a **Refutation Process.**

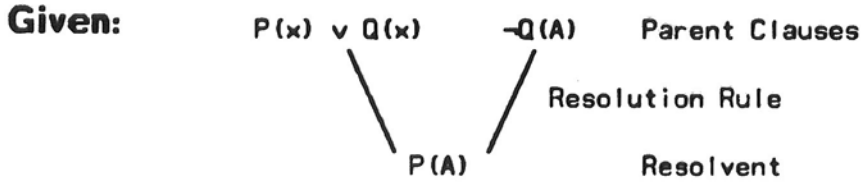
The key idea is to compute the *resolvent* of two *parent* clauses.

> [A clause is a disjunction of literals in which all variables are universally quantified. We will see an algorithm for converting any PC wff into a set of clauses.]

The resolvent is another clause that logically follows from its parents.

> Resolution is a generalization of: $[(A \vee B) \wedge (C \vee \neg B)] \rightarrow (A \vee C)$

Example:



We need to consider two processes:

1.) Conversion to clause form.

2.) Matching and unification of literals – where unification means finding substitutions for variables that make expressions identical. (A general form of **pattern matching.**)

Conversion to Clause Form

Steps:

1.) Eliminate implication signs.

2.) Reduce scopes of negation signs. [Negate only one predicate]

3.) Standardize variables.

4.) Eliminate existential quantifiers. [Skolem functions]

5.) Convert to prenex form. [Prefix/Matrix]

6.) Put matrix in conjunctive normal form.

7.) Eliminate universal quantifiers.

8.) Eliminate $\wedge$ signs.

Example:

$(\forall x)(\forall y)\{[A(x) \rightarrow \neg C(x,y)] \rightarrow \neg(\forall x)(\exists z)[P(x,z) \wedge R(z)]\}$

Step 1
$(\forall x)(\forall y)\{\neg[\neg A(x) \vee \neg C(x,y)] \vee \neg(\forall x)(\exists z)[P(x,z) \wedge R(z)]\}$
Step 2
$(\forall x)(\forall y)\{\neg[\neg A(x) \vee \neg C(x,y)] \vee (\exists x)\neg(\exists z)[P(x,z) \wedge R(z)]\}$
$(\forall x)(\forall y)\{\neg[A(x) \vee \neg C(x,y)] \vee \neg(\exists x)(\forall z)\neg[P(x,z) \wedge R(z)]\}$
$(\forall x)(\forall y)\{\neg[\neg A(x) \vee \neg C(x,y)] \vee (\exists x)(\forall z)[\neg P(x,z) \vee \neg R(z)]\}$
$(\forall x)(\forall y)\{[A(x) \wedge C(x,y)] \vee (\exists x)(\forall z)[\neg P(x,z) \vee \neg R(z)]\}$
Step 3
$(\forall x)(\forall y)\{[A(x) \wedge C(x,y)] \vee (\exists u)(\forall z)[\neg P(u,z) \vee \neg R(z)]\}$
Step 4
$(\forall x)(\forall y)\{[A(x) \wedge C(x,y)] \vee (\forall z)[\neg P(g(x,y),z) \vee \neg R(z)]\}$
Step 5
$\underline{(\forall x)(\forall y)(\forall z)}\{[A(x) \wedge C(x,y)] \vee [\neg P(g(x,y),z) \vee \neg R(z)]\}$
Step 6
$(\forall x)(\forall y)(\forall z)\{[A(x) \vee \neg P(g(x,y),z) \vee \neg R(z)] \wedge [C(x,y) \vee \neg P(g(x,y),z) \vee \neg R(z)]\}$
Step *7*
$[A(x) \vee \neg P(g(x,y),z) \vee \neg R(z)] \wedge [C(x,y) \vee \neg P(g(x,y),z) \vee \neg R(z)]$
Step 8
$A(x) \vee \neg P(g(x,y),z) \vee \neg R(z)$
$C(x,y) \vee \neg P(g(x,y),z) \vee \neg R(z)$

Unification

Substitution: s = $\{t_1/v_1, t_2/v_2, \ldots\}$

Means substitute t_i for v_i, where t_i is a term, and v_i is a variable.

Examples

Given: L = P(x,f(y),B)

s_1 = {z/x, z/y} Ls_1 = P(z,f(z),B)

s_2 = {f(A)/x, B/y} Ls_2 = P(f(A),f(B),B)

Given: L = IN(?x,?y)

s = {BLOCK1/?x,BLOCK2/?y} Ls = IN(BLOCK1,BLOCK2)

Definition: Given two literals, L_1 and L_2, we say that substitution s *unifies* L_1 and L_2 provided $L_1s = L_2s$.

Examples

L1 = Q(x,f(y),z) L2 = Q(B,f(g(A)),z)

s = {B/x, g(A)/y}

s = {B/x, g(A)/y, A/z)

There exists an algorithm for finding the meet general unifier (MGU) for a set of literals. It is based on pairwise unification.

Example

S = {P(x,y), P(f(z),x), P(w,f(A))}

S_1 = {P(x,y), P(f(z),x)}

σ_{s_1} = {f(z)/x, f(z)/y}

$S_2 = \{S_1\sigma_{s_1}, P(w,f(A))\} = \{P(f(z),f(z)), P(w,f(A))\}$

σ_{s_2} = {f(A)/w, A/z}

$\sigma_s = \sigma_{s_1}\sigma_{s_2}$ = {f(A)/x, f(A)/y, f(A)/w, A/z}

$S\sigma_s$ = {P(f(A), f(A))}

Unification

```
(UNIFY
  (LAMBDA (E1 E2)
    (PROG (TEMP Z1 Z2)
          (COND
            ((OR (ATOM E1) (ATOM E2))
              (COND
                ((NOT (ATOM E1)
                  (SETQ TEMP E1)
                  (SETQ E1 E2)
                  (SETQ E2 TEMP)))
              (COND
                ((EQ E1 E2)
                  (RETURN NIL))
                ((VARIABLE E1)
                  (COND
                    ((OCC E1 E2) (RETURN 'FAIL))
                    (T (RETURN (LIST (CONS E2 E1))))))
                ((VARIABLE E2)
                  (RETURN (LIST (CONS E1 E2))))
                (T (RETURN 'FAIL)))))
          (SETQ Z1 (UNIFY (CAR E1) (CAR E2)))
          (COND
            ((EQ Z1 'FAIL) (RETURN 'FAIL)))
          (SETQ Z2 (UNIFY (SUBSTITUTE Z1 (CDR E1))
                          (SUBSTITUTE Z1 (CDR E2))))
          (COND
            ((EQ Z2 'FAIL) (RETURN 'FAIL)))
          (RETURN (COMPOSE Z1 22)))))

(UNIFY ‘(P x) ‘(P A))
  ((A . x))
    (P A)

(UNIFY ‘(P (f x) y (g y)) ‘(P (f x) z (g x)))
  ((x . y) (x . z))
    (P (f x) x (g x))

(UNIFY ‘(P (f x (g A y)) (g A y)) ‘(P (f x z) z))
  (((g A y) . z))
    (P (f x (g A y)) (g A y))
```

Resolution

Forming Resolvents of Pairs of Clauses

Given two parent clauses, $\{L_i\}$ and $\{M_i\}$, with no common variables.

Find subsets $\{l_i\} \subseteq \{L_i\}$ and $\{m_i\} \subseteq \{M_i\}$ such that MGU λ exists for $\{l_i\} \cup \{\neg m_i\}$.

Example:

$\{L_i\} = P(x,f(A)) \vee P(x,f(y)) \vee Q(y)$
$\{M_i\} = \neg P(z,f(A)) \vee \neg Q(z)$

One possibility:

$\{l_i\} = \{P(x,f(A))\}$
$\{mi\} = \{\neg P(z,f(A))\}$
$\lambda = \{z/x\}$

Resolvent:

$[\{L_i\} - \{li\}]_\lambda \cup [\{Mi\} - \{mi\}]_\lambda = P(z,f(y)) \vee Q(y) \vee \neg Q(z)$

Key Idea: If two clauses are *unifiable* and their resolvent is NIL, then they are contradictory – not simultaneously satisfiable.

Example:

$P(A)$ and $\neg P(z)$ resolve to NIL under $\lambda = \{A/z\}$

Resolution Refutation

Example:

Given:

1. $(\forall x)(\exists y)\{MAILMAN(x) \rightarrow [DOG(y) \wedge HAS\text{-}BITTEN(y,x)]\}$
2. $MAILMAN(OSCAR)$
3. $(\forall x)(\forall y)[HAS\text{-}BITTEN(x,y) \rightarrow \neg FRIENDS(y,x)]$

Prove:

4. $(\exists z)[\neg FRIENDS(OSCAR,z)]$

Negate 4

5. $\neg(\exists z)[\neg FRIENDS(OSCAR,z)]$

Put 1 in clause form

$(\forall x)(\exists y)\{\neg MAILMAN(x) \vee [DOG(y) \wedge HAS\text{-}BITTEN(y,x)]\}$
$(\forall x)\{\neg MAILMAN(x) \vee [DOG(f(x)) \wedge HAS\text{-}BITTEN(f(x),x)]\}$

Distributive Law: $(A \vee (B \wedge C)) \rightarrow (A \vee B) \wedge (A \vee C)$

$(\forall x)\{[\neg MAILMAN(x) \vee DOG(f(x))] \wedge [\neg MAILMAN(x) \vee HAS\text{-}BITTEN(f(x),x)]\}$

6. $\neg MAILMAN(x) \vee DOG(f(x))$
7. $\neg MAILMAN(x) \vee HAS\text{-}BITTEN(f(x),x)$

Put 3 in clause form

8. $\neg HAS\text{-}BITTEN(z,y) \vee \neg FRIENDS(y,z)$

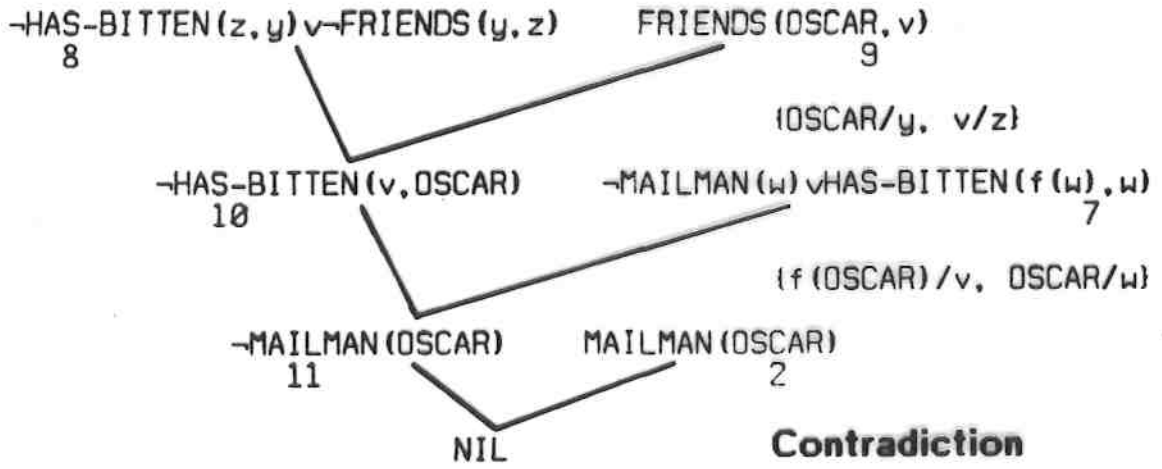
Put 5 in clause form

5. $\neg(\exists z)[\neg FRIENDS(OSCAR,z)]$
   $(\forall v)[FRIENDS(OSCAR,v)]$

9. $FRIENDS(OSCAR,v)$

Resolution Refutation

Start resolving



Base Set: 2,6,7,8,9
8 and 9 are the *Parents* of 10 – it is their *Descendant*
Derivation Graph
Refutation Tree

Production System:
Database: Set of Clauses
Rule: Resolution
Control Strategy: Irrevocable (Commutative)

Possible Control Strategies:

1. Breadth-First [Complete]

Restricting Resolutions:

2. Set-of-Support (negated goal wff or descendants) [Complete]
3. Linear-Input Form (base set) [Incomplete]
4. Ancestry-Filtered (base set or parent-ancestor) [Complete]

Ordering Strategy:

5. Unit Preference (single-literal clauses)

Answer Extraction

Suppose we do not simply want to prove a theorem, but also to extract some answer. With exactly what z is OSCAR not on friendly terms? We desire *a constructive proof.*

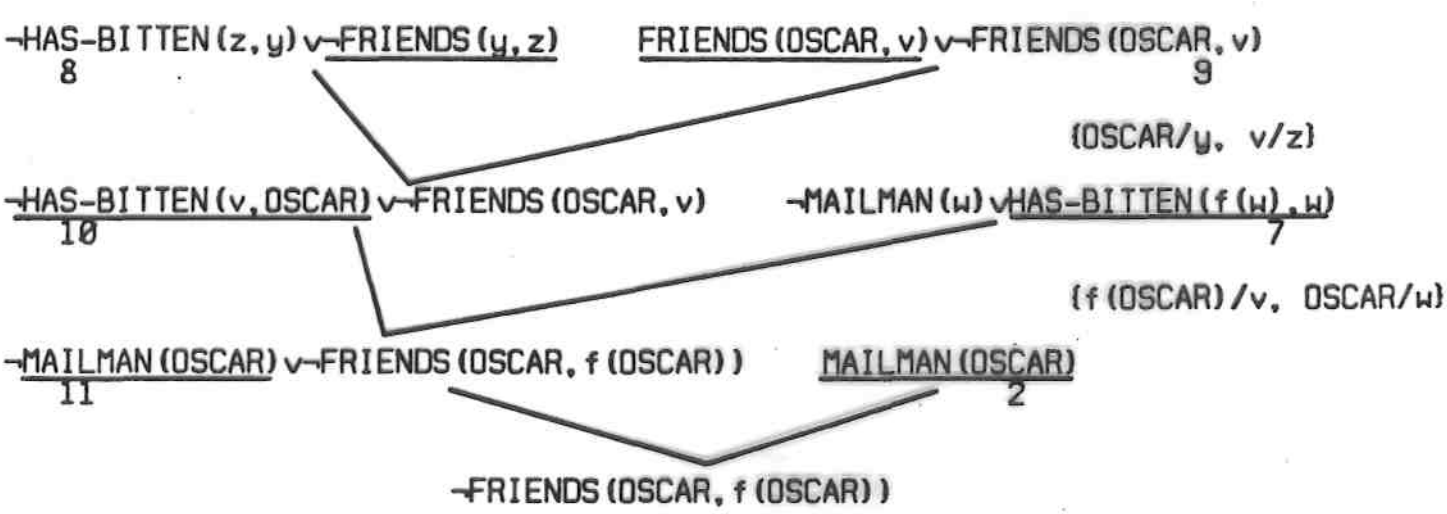
Method:

1) Append to each clause arising from the negation of the goal wff its own negation.
   FRIENDS(OSCAR,v) → FRIENDS(OSCAR,v) ∨ ¬FRIENDS(OSCAR,v)

2) Perform the same resolutions as used before in deriving a contradiction.

3) The clause left at the root *is* the answer statement.

Example Revisited:



What is f(OSCAR)? A Skolem function arising from the first wff. f(OSCAR) is the domain element such that:
DOG(f(OSCAR)) ∧ HAS-BITTEN(f(OSCAR),OSCAR) = T

Rule-Based Deduction Systems

What did we lose by going to clause form? ... Extra-logical, domain-specific, control information.

Rule-Based Deduction: Use *direct* methods – not *refutation* methods. Leave implications in their original form (Efficiency).

Rules: General knowledge about a problem domain – including *control* knowledge.

Facts: Specific knowledge relevant to a particular case.

[Resolution theorem-prover as a production system]

Some rules are best applied in the forward direction – some in the backward direction. Search in the direction of decreasing number of alternatives.

$CAT(x) \rightarrow ANIMAL(x)$ *forward*

$\neg ANIMAL(x) \rightarrow \neg CAT(x)$ *backward*

Control Knowledge:

1) Direction of rule application.
2) Order in which to try subgoals.
   $[P1 \wedge P2 \wedge P3] \rightarrow Q$
3) Which rules to apply to a particular subgoal.
4) Order in which to try rules.
5) Which rules to apply after solving a particular subgoal (*demons*).

Placement of Control Knowledge:

1) *Object-level* Knowledge vs *Meta-level* Knowledge.
2) Rules as programs. Procedural Representation (e.g., PLANNER).

Robot Problem Solving

Simple environment – “blocks” world.

Synthesis of a sequence of robot actions that (if executed correctly) will achieve a stated goal – *planning* as opposed to *plan execution.*

Commutative Production Systems (Resolution): How to handle rules that must *delete* wffs when applied?

Could use *situation* variable in each predicate.

$ON(C,A,S_0) \wedge CLEAR(C,S_0) \wedge CLEAR(B,S_0)$

Prove: $(\exists s)[ON(A,B,s) \wedge ON(B,C,s)]$

An action is a function that maps one situation into another.

`place(x,y,s):`

$[CLEAR(x,s) \wedge CLEAR(y,s) \wedge DIFF(x,y)] \rightarrow ON(x,y,place(x,y,s))$

Frame Problem: What about assertions that are *not* affected by actions?
Need **frame axioms.**

$[ON(u,v,s) \wedge DIFF(u,x)] \rightarrow ON(u,v,place(x,y,s))$

QA3 – Cordell Green. One frame axiom per unaffected predicate per action. (Can do better than this by treating relations as individuals [*on(A,B)*] or by using higher-order logic.)

Combinatorial Explosion. Many frame assertions. Control ...

STRIPS

Model actions as standard production rules.

Precondition: conjunction of literals

Action: **add list + delete list**

```
place(x,y):

  precondition:    CLEAR(y) ∧ HOLDING(x) [ ∧ DIFF(x,y)]
  action:
    add list:      ON(x,y) ∧ HANDEMPTY
    delete list:   CLEAR(y) ∧ HOLDING(x)


grasp(x):

  precondition:    CLEAR(x) ∧ HANDEMPTY
  action:
    add list:      HOLDING(x)
    delete list:   HANDEMPTY
```

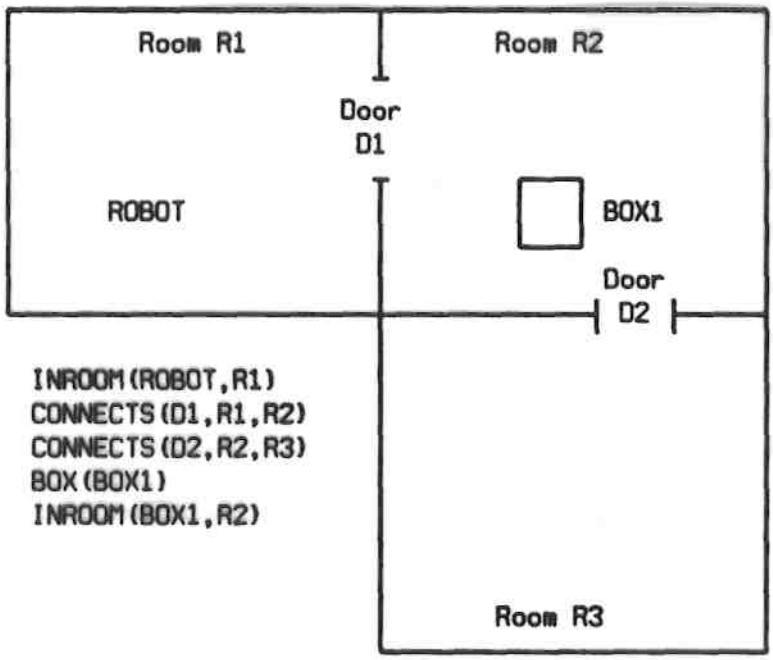
Don't need frame axioms. Explicitly indicate assertions that are changed by an action. This assumes ... ?

Forward Reasoning: Find literals among the facts that unify with each of the precondition literals in a consistent way. If such literals can be found, then the rule precondition *matches* the facts and the rule can be applied.

Backward Reasoning: Find a literal contained in the goal description that unifies with one of the literals in the add list. If such literals can be found, then the rule can be applied.

STRIPS

Example:



$(\forall x)(\forall y)(\forall z)[CONNECTS(x,y,z) \rightarrow CONNECTS(x,z,y)$

Goal wff

$(\exists x)[BOX(x) \wedge INROOM(x,R1)]$

GOTHRU(d,r1,r2):

pre:	INROOM(ROBOT,r1) $\wedge$ CONNECTS(d,r1,r2)
add:	INROOM(ROBOT,r2)
delete:	INROOM(ROBOT,$)

PUSHTHRU(b,d,r1,r2):

pre:	INROOM(b,r1)$\wedge$INROOM(ROBOT,r1)$\wedge$CONNECTS(d,r1,r2)
add:	INROOM(ROBOT,r2), INROOM(b,r2)
delete:	INROOM(ROBOT,$), INROOM(b,$)

The GPS Algorithm

GPS searches forward by iterating through a cycle of difference measurement and operator application. Recursion is used whenever the an operator cannot be applied because of unsatisfied preconditions. Depth-first backtracking is used when faced with dead-end situations.

```
(GPS
  (LAMBDA (S G O)
    (PROG (D Q P S1)
     LOOP (COND
            ((MATCH S G) (RETURN S)))
     BTP1 (SETQ D (FIND-DIFFERENCE S G))
          (COND
            ((NULL D) (RETURN 'FAIL)))
     BTP2 (SETQ Q (FIND-OPERATOR D O S))
          (COND
            ((NULL Q) (GO BTP1)))
          (SETQ P (PRECONDITION Q D S))
          (SETQ S1 (GPS S P O))
          (COND
            ((EQ S1 'FAIL) (GO BTP2)))
          (SETQ S (APPLY-OPERATOR Q S1))
          (GO LOOP))))
```

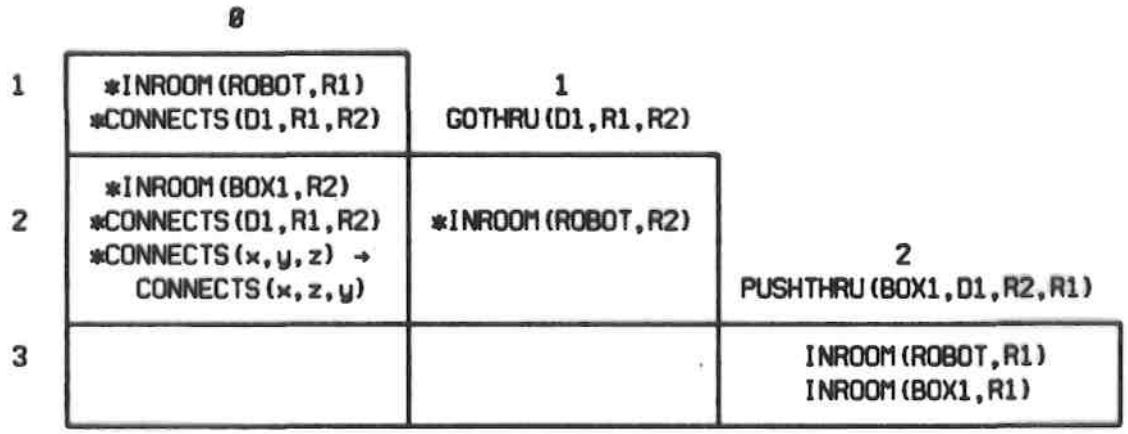
Heuristics for Controlling Search

1) Each subgoal in a sequence should be easier than the preceding subgoal.
2) A goal should be easier than all of its ancestors.
3) A new object should not be much larger than the objects in the top-level goal.
4) Once a goal has been generated, an identical goal should not be generated again.
5) Limit the total depth of the search tree.

STRIPS

STRIPS uses means-ends analysis (GPS).

Triangle Tables



Cell (i,0) contains clauses from the original model that are still true when rule i is to be applied and that are preconditions of rule i.

Marked clauses elsewhere in row i are preconditions for rule i that are added to the model by previous rules.

The effects of applying rule i are shown in row i+1. The rule's add list appears in cell (i+1,i). For each previous rule, say, rule j, clauses added by rule j and not yet deleted are copied into cell (i+1 ,j).

The add list for a sequence of rules 1 through i, taken as a whole, is given by the clauses in row i+1 (excluding column 0).

The preconditions for a sequence of rules i through n, taken as a whole, are given by the marked clauses in the rectangular subarray containing row i and cell (n+1,0). This rectangle is called the i^{th} *kernel* of the plan.

Difficulties with STRIPS and Means-Ends Analysis

1. Non-optimal plans.
2. Interacting solutions to subgoals.
3. Limitations in representing world (Frame Problem).
4. Combinatorial problems.

ABSTRIPS

Key Ideas:

1. Recognize the most significant features of a problem.
2. Plan an outline of a solution in terms of those features.
3. Plan at progressively greater levels of detail (top-down design).

Method:

Attach *criticality value* to each literal. Ignore literals in preconditions that are below current level of criticality.

Assigning Criticality Values to Literals: two Steps: manual start – automatic refinement.

General Guiding Criteria:

1) If the truth value of a literal cannot be changed by any rule, it gets maximum criticality
2) If a literal L in a particular rule can easily be achieved once other preconditions of the same rule are satisfied, then L should be less critical than those others.
3) If satisfying L requires conditions beyond those others in the rule precondition, then L gets high, but not maximum, criticality.

Example:

TURN-ON-LAMP(x):

pre: TYPE(x,LAMP) ∧ (∃r)[INROOM(ROBOT,r) ∧
INROOM(x,r)] ∧ PLUGGED-IN(x) ∧ NEXTTO(ROBOT,x)

Step 1: Initial Assignment

Predicate	Rank
TYPE	4
INROOM	3
PLUGGED-IN	2
NEXTTO	1

ABSTRIPS

Step 2: Refinement

1) TYPE cannot be changed by any available rule → highest criticality (6).
2) INROOM appears in add list of some rule, but requires more conditions than TYPE, so it gets high criticality, but not maximum, value → 5.
3) PLUGGED-IN can be achieved by a plan based on the above preconditions. Therefore its value → 2.
4) NEXTTO – ditto above, value → 1.

Control of ABSTRIPS (recursive function):

Form plan at highest level of abstraction and refine. Input: criticality level and plan (list of rules) to be refined

Check plan by seeing whether each rule applies. If not, set up subgoal and solve *at the current level of criticality*

If subgoal cannot be solved, reject plan and return control to a more abstract level. Reject failed node and form alternate plan.

Results:

11 rule plan: STRIPS: 30 CPU-min, ABSTRIPS: 5 CPU-min, 50% of nodes expanded

Natural Language Processing

Why Study?

Easier and more natural communication with computers

Method for extending our understanding of language, understanding, and mind.

Difficulties:

Sheer size and complexity of grammar and vocabulary – artificial vs natural languages

Ambiguity, pronouns, ellipsis, ungrammatical utterances.

"Time flies like an arrow"

"Bill hit Joe on the shoulder"
"He was black and blue by the next morning"

"I saw the astronomer on the hill with a telescope"

Large amount of common-sense knowledge needed (even for two sentence children's stories)

Representing meaning of words and sentences (1st order predicate calculus insufficient)

What is *meaning?* What is *understanding?*

Natural Language Processing

Grammars:

1. Context-free (type 2)

```
S → NP VP          DET → the
NP → DET NOUN      NOUN → boys
NP → NOUN          NOUN → apples
VP → VERB NP       VERB → eat
```

Parsing:

```
              S

         NP          VP

 DET   NOUN       VERB   NP

 the   boys        eat  NOUN

                        apples
```

Unable to extend context-free grammars to capture natural language complexity.

Chomsky: deep structure + *transformations* → surface structure

Analysis cannot be done in two separate and sequential phases: syntactic and semantic.

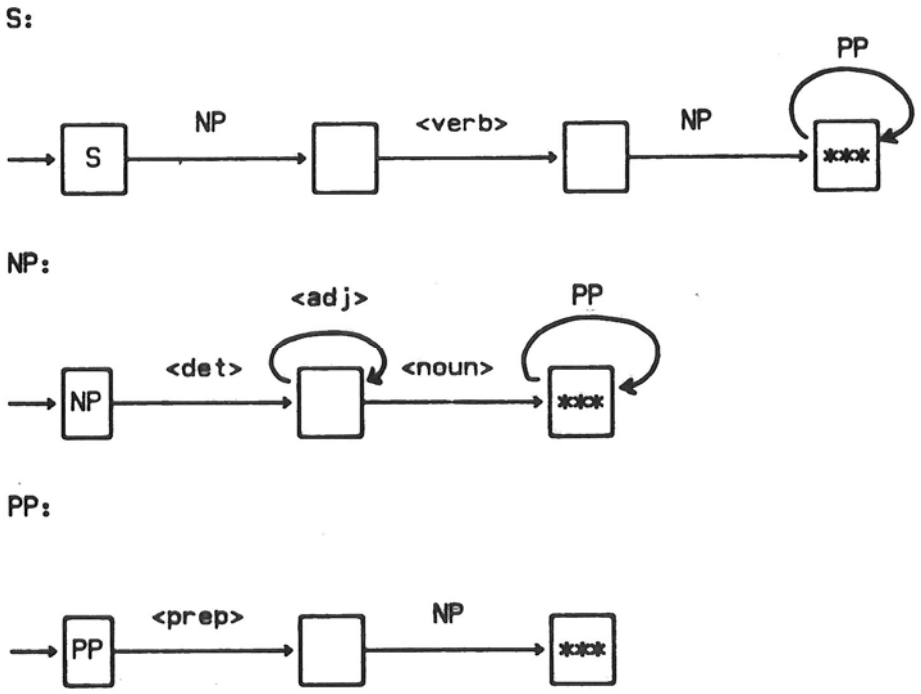
Case Grammars: relationships between noun phrases and verb (e.g., agent – instigator of action, object)

Augmented Transition Networks

A representation for natural language grammars. Finite State Transition Diagrams, Recursive Transition Networks, → ...

Arcs (terminal or non-terminal symbols of the grammar): the non-terminal symbols imply recursion

Nodes: states (START → FINAL)



The little boy in the swimsuit kicked the red ball.

ATN:

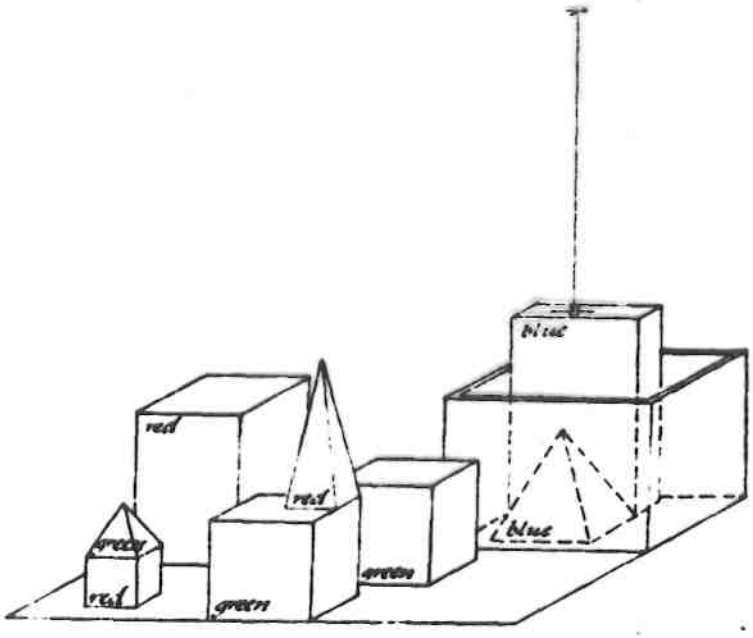
1) Registers to store information (e.g., partial derivation trees).
2) Arbitrary tests on arcs (e.g., semantic checks).
3) Actions attached to arcs (e.g., to modify data structure returned).

Widely-used for constructing different types of grammar for question-answering systems, text generation systems, and speech-understanding systems.

Pick up a big red block.
OK.
Grasp the pyramid.
I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.
Find a block which is bigger than the one you are holding and put it into the box.
BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I AM HOLDING.
OK.
What does the box contain?
THE BLUE PYRAMID AND THE BLUE BLOCK.

....

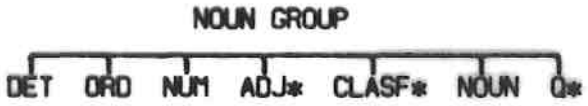
Had you touched any pyramid before you put the green one on the little cube?
YES, THE GREEN ONE.
When did you pick it up?
WHILE I WAS STACKING UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE.
Why?
TO GET RID OF IT.
Why did you do that?
TO CLEAN OFF THE RED CUBE.
Why did you do that?
TO STACK UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE.
Why did you do that?
BECAUSE YOU ASKED ME TO.



SHRDLU

Terry Winograd 1971: Simple blocks world dialogues, small vocabulary, written in Micro-Planner, covers total problem of natural language understanding.

1) Syntactic Parser: Uses *systemic grammar* – central items are groups of words with slots for various features: (Note the reason for dealing with syntax).




```
DET: the   ORD: first    NUM: three   ADJ: old
CLASF (classifier): city CLASF: fire  NOUN: hydrants
Q (qualifier): PREPG: without covers  Q: CLAUSE: you can find
```

Parser is top-down, left-right with some *demons* to change flow (e.g., recognizing an "AND" – conjoined structures). Constrained Backup to deal with ambiguity – try to get it right the first time.

2) Semantic Analysis: Dictionary includes semantic information (properties) (e.g., table is immovable). Word definitions are programs to be executed. (Note similarity to ATN's)

3) Pragmatics or Discourse Knowledge: local context (e.g., to answer "Why" questions), global context (e.g., "Pick up the pyramid" is ambiguous), world knowledge.

Integrated syntax/semantics/pragmatics + narrow domain of discourse helps to avoid backup.

Translating sentences into programs to be executed in response to an imperative.

SHRDLU

Limitations:

1) Ad hoc representation of speaker/hearer internal structure: hard to extend

2) Word definition as program is still inadequate: in general, words have definitions only in context (e.g., Is the Pope a bachelor).

3) Thinking of every utterance as a command to do something is inadequate. The hearer is engaged in a complex process of trying to understand what the speaker is saying and why.

4) Representation and reasoning operations inadequate for common-sense knowledge normally seen in language. Deductive logic doesn't cover everything.

Speech Understanding

Isolated Word Recognition (1960's): Compare signal to templates (a priori stored representations of the acoustical pattern of words in vocabulary). Select best match according to *distance* metric. Problems: noise, differing pronunciation from time to time by a single speaker, by multiple speakers. Current systems: 120 words, 99.5% accuracy (tuning vs vocabulary size). Cost <$80k.

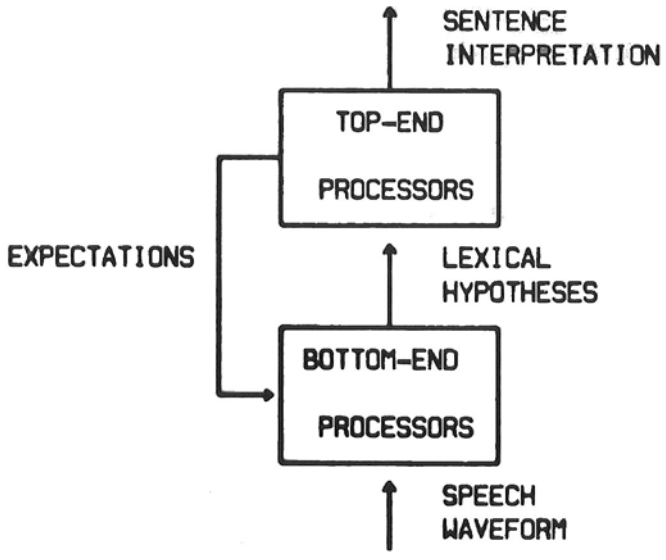
Problems with Connected Speech:

1) Pronunciation of individual words changes when words are juxtaposed to form a sentence – syllables are dropped ("swallowed") at word boundaries.

2) Word boundaries are hard to find.

The signal doesn't look like a concatenation of the signals associated with individual words. Combinatorial explosion again!

How to proceed: Use syntactic and semantic knowledge to generate *expectations* about the content of an utterance – to reduce search.



Speech Understanding

ARPA Project (1971-76):

GOALS:

1) Accept connected speech
2) From many, cooperative speakers (~5)
3) In a quiet room with good microphone
4) With a few training sentences per speaker
5) Accepting a 1000 word vocabulary
6) Using an artificial syntax (finite-state language, restricted ATN)
7) In a constraining task (document retrieval, travel management)
8) Yielding < 10% semantic error
9) In a few times real time (on a 100 MIPS computer).

Sources of Knowledge:

1) *Phonetics:* representations of the physical characteristics of the sounds in all words in the vocabulary

2) *Phonemics:* rules describing variations in pronunciation that appear when words are spoken together in sentences (co-articulation across word boundaries, "swallowing" of syllables)

3) Morphemics: rules describing how morphemes (units of meaning) are combined to form words (formation of plurals, verb conjugations, etc.)

4) Prosodics: rules describing fluctuation of stress and intonation across a sentence.

5) Syntax: grammar

6) Semantics: meaning of words and sentences

7) Pragmatics: rules of conversation

Lexicon: Dictionary – represented internally in terms of the pronunciations of all words.

HEARSAY-II

Document Retrieval Task: “Are any by Feigenbaum and Feldman?”

Results: 90% semantic accuracy, 73% word-for-word correct, 1011 word vocabulary, 60-80 MIPS for real-time response. Highly-constrained grammars (BF = 33, 46). Grammar stylized for task: noun → topic | word | author | ...

Ideas:

Blackboard (Global Database): to simplify flexible cooperation. Contains hypotheses at multiple levels of abstraction. Uniform structure for all hypotheses – connected through AND/OR graph. Idea of “support from below” for a hypothesis.

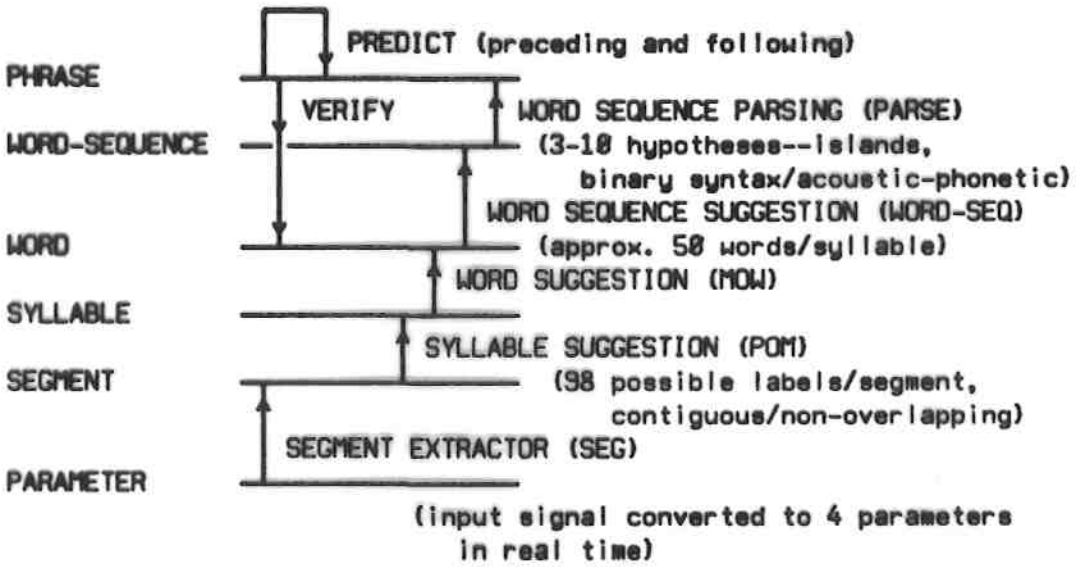
Cooperating Experts: Multiple knowledge sources (KSs). (The rules in a production system.) Separate, anonymous, asynchronous, and independent – to increase flexibility – addition, deletion, modification. A KS has a PRECONDITION (a test on one hypothesis level) and an ACTION.

Hypothesize-and-Test: All cooperation (KS ACTIONs) viewed as creation, testing, and modification of hypotheses about the utterance. Basic assumption: KSs make errors. The control structure has to take this into account.

Control Strategy: KSs are triggered by a *blackboard monitor.* KSs are scheduled according to priorities calculated from *stimulus frame* (set of hypotheses appropriate for action by a KS) and *response frame* (stylized description of the likely action of a KS). (Note: Asynchronous processing down to word level only – why?)

HEARSAY-II

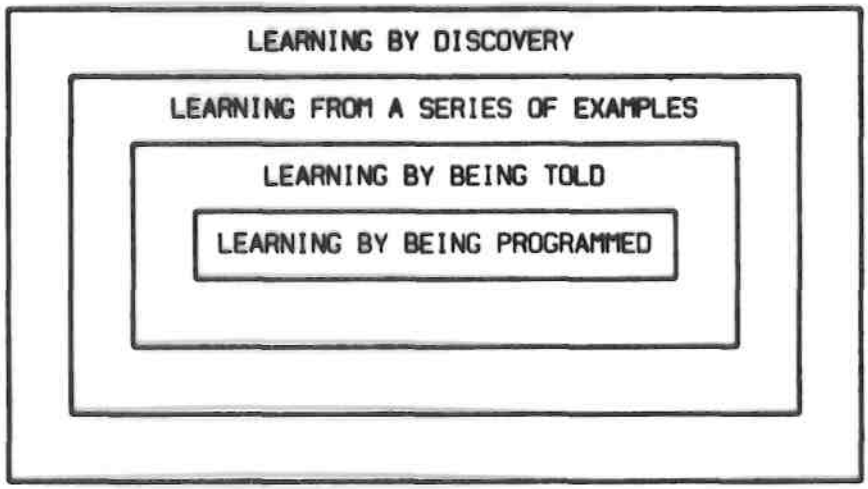
Blackboard Partitioning/Scope of KS Operation:



MACHINE LEARNING

Learning System: A system that uses information obtained during one interaction with its environment to improve its performance during subsequent interactions.

Types of Learning:



Approaches to Learning Systems:

Adaptive Control: Estimation of parameters of a mathematical structure chosen by the designer to represent either the system to be controlled or the controller. Stability, Convergence [provable].

[Assumes that the world can be modeled parametrically.]

Pattern Recognition: Learn rules for assigning patterns (usually feature vectors) to particular known classes by examining a set of patterns with known class membership. Convergence, Risk, Optimality, [provable].

[Assumes that patterns can be modeled as points in n-space, and classification (finding a set of decision surfaces) is the problem.]

Artificial Intelligence: Learn structural (symbolic) descriptions for patterns, concepts. Use of domain-specific knowledge in addition to general-purpose inductive mechanisms. Knowledge Representation (networks, rules,...), Generalization Languages, Matching, Search, Complexity.

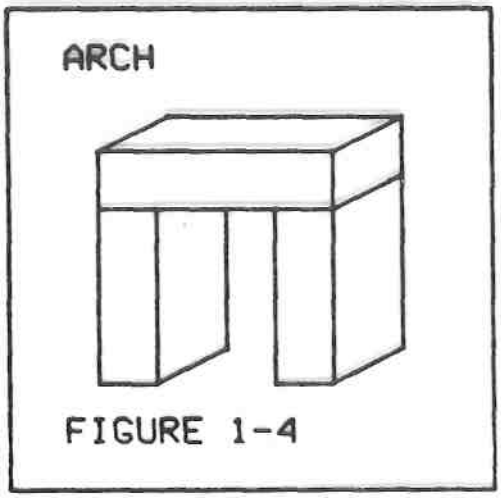


FIGURE 1-4

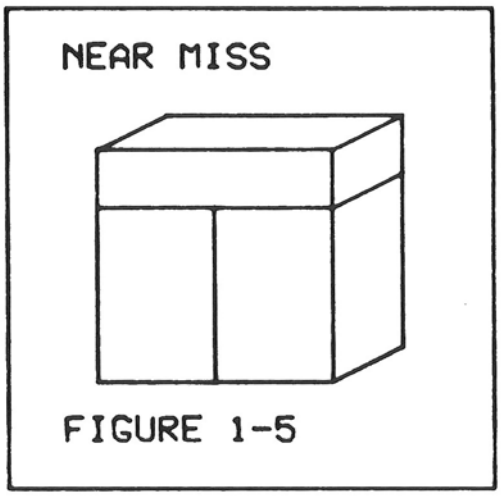


FIGURE 1-5

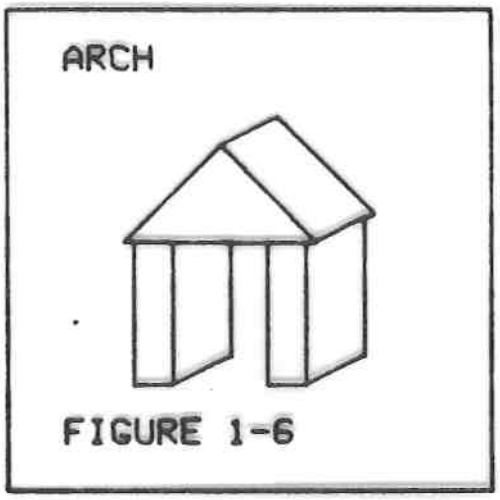


FIGURE 1-6

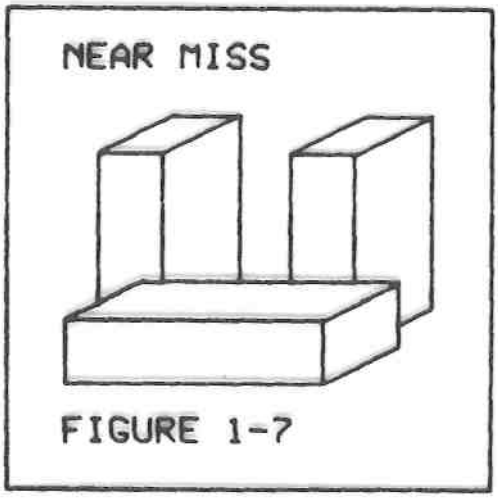
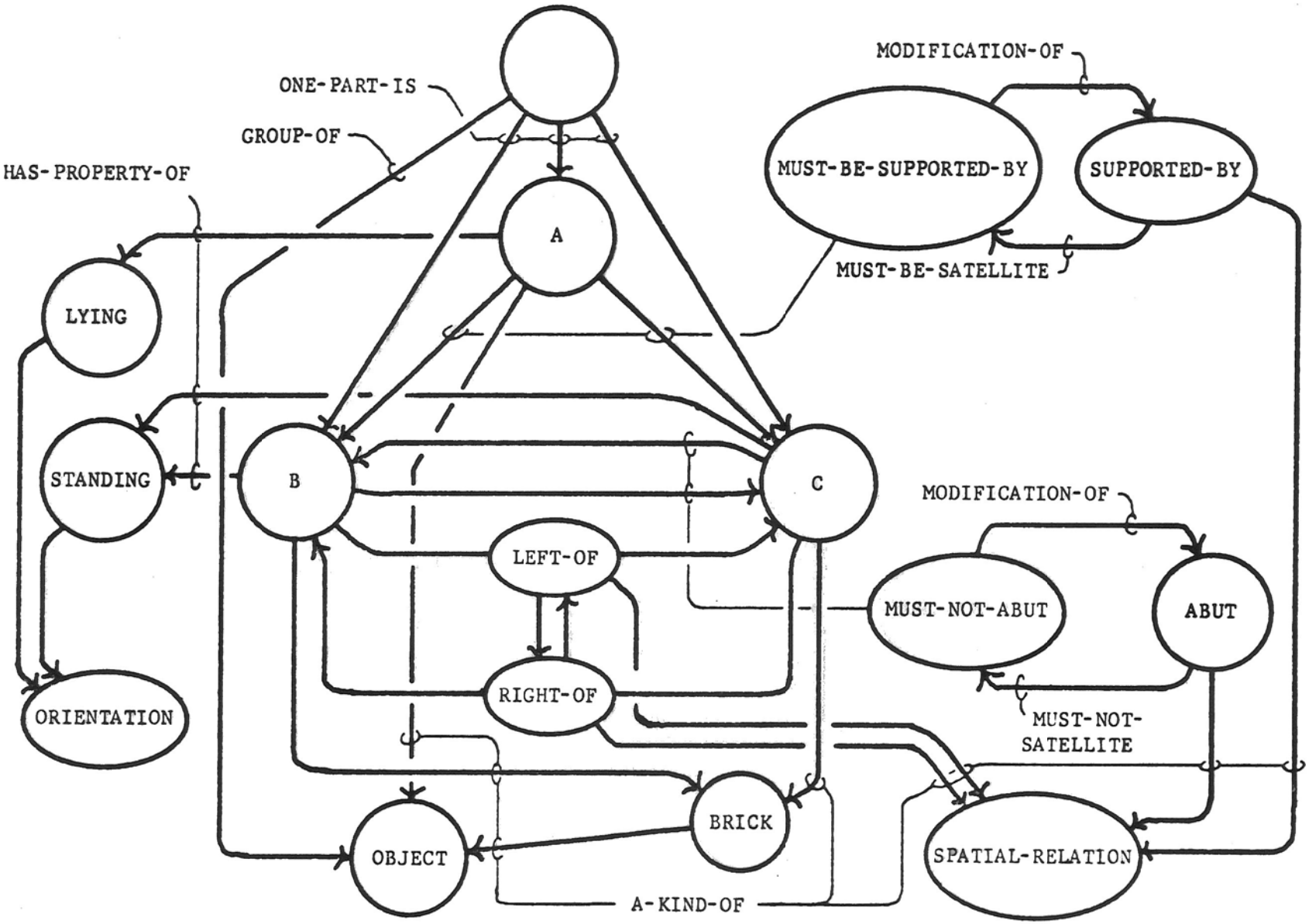


FIGURE 1-7


ONE-PART-IS
GROUP-OF
HAS-PROPERTY-OF
A
B
C
LYING
STANDING
ORIENTATION
LEFT-OF
RIGHT-OF
OBJECT
BRICK
A-KIND-OF
MODIFICATION-OF
MUST-BE-SUPPORTED-BY
SUPPORTED-BY
MUST-BE-SATELLITE
MODIFICATION-OF
MUST-NOT-ABUT
ABUT
MUST-NOT-SATELLITE
SPATIAL-RELATION

MACHINE LEARNING

Checkers: (Samuel: late 50's, early 60's)

1) **Rote Learning:** When you encounter a position for the first time, store its description and static value estimate. In a future game, if you encounter the position (say during look-ahead), use the previously stored static value. This has the effect of extending the number of ply that you look ahead. [Slow, continuous learning rate: Most effective during opening and end-game phases of play. Why?]

2) **Parameter Learning:** Modify the static evaluation polynomial.

   $V(p) = c_1f_1(p) + c_2f_2(p) + ... + c_nf_n(p)$

   How to know that “correct” score of a position?

   1) From *Book Games:* Compare machine's ranking of positions to expert's ranking.
   2) *Compare static value with backed-up value:* Assume consistency.

   Problems: Assumption of linear polynomials (no interaction between terms), No method for generating new parameters. [Mesa phenomenon]

   Samuel later considered interactions – with significant improvement in performance [Signature Tables].

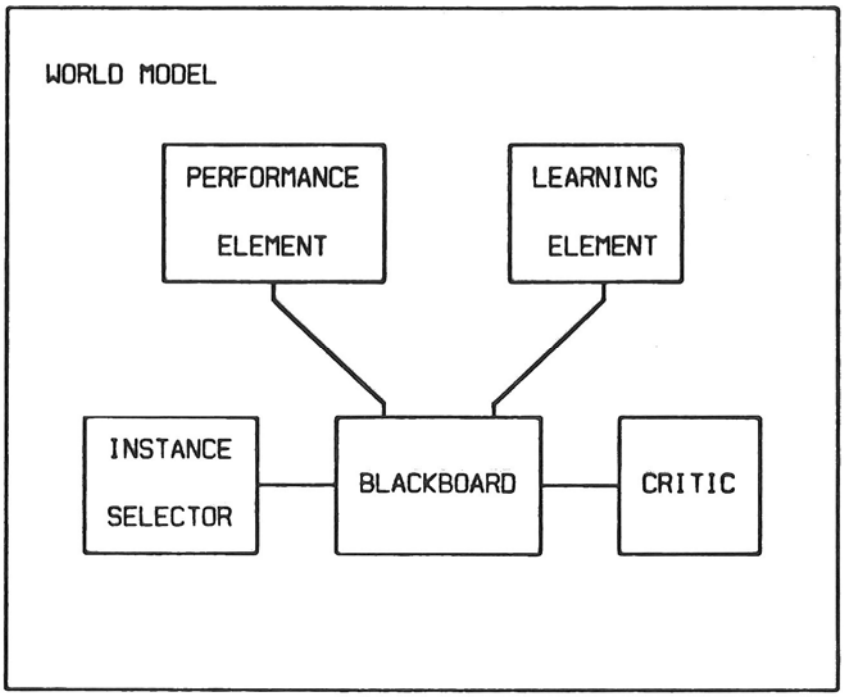
   Samuel's program achieved world championship caliber.

STRIPS and **MACROPS**: *Method Learning*

Winston, 1970: *Concept Learning*

MACHINE LEARNING

Learning System Model:



BLACKBOARD: Global Database (as per HEARSAY-II)

PERFORMANCE ELEMENT: Problem Solver (e.g., MYCIN)

INSTANCE SELECTOR: Supplies examples and non-examples of concept to be learned (+ve and -ve training instances) [Teacher].

CRITIC: Evaluation, Localization (*credit assignment*), and Recommendation (what *kind* of change to make – not how to *implement* that change).

LEARNING ELEMENT: Interface between Critic and Performance Element. Abstract Recommendations → Specific Changes.

WORLD MODEL: Conceptual Framework. Definitions of objects, methods, etc. The type of information that can be learned.

[Systems inside of Systems]

GENERALIZATION AS SEARCH

Learning from a set of training instances (as per Winston

Given:

1) **Instance Language**

   Unordered pairs of objects, characterized by three properties

Size:	`Large, Small`
Color:	`Red, Orange, Yellow`
Shape:	`Square, Circle, Triangle`

   Example: `{ (Large Red Square) (Small Yellow Circle) }`

2) Generalization Language

   Example: `{ (Small ? Circle) (Large ? ?) }`

3) **Matching Predicate:** Tests whether an instance and a generalization *match* (i.e., whether the instance is contained in the instance set that is delimited by the generalization).

4) **Training Instances:** Each is an instance from the given language, along with its classification *as* either an instance of the target generalization (*positive instance*) or not an instance of the target generalization *(negative instance*)

Determine: Generalizations within the given language that are *consistent* with the training instances (i.e., plausible descriptions of the target generalization)

The generalization language describes a *hypothesis space* (search space) of possible generalizations.

More-Specific-Than Relation: A partial ordering of the hypothesis space-gives a method for organizing the search.

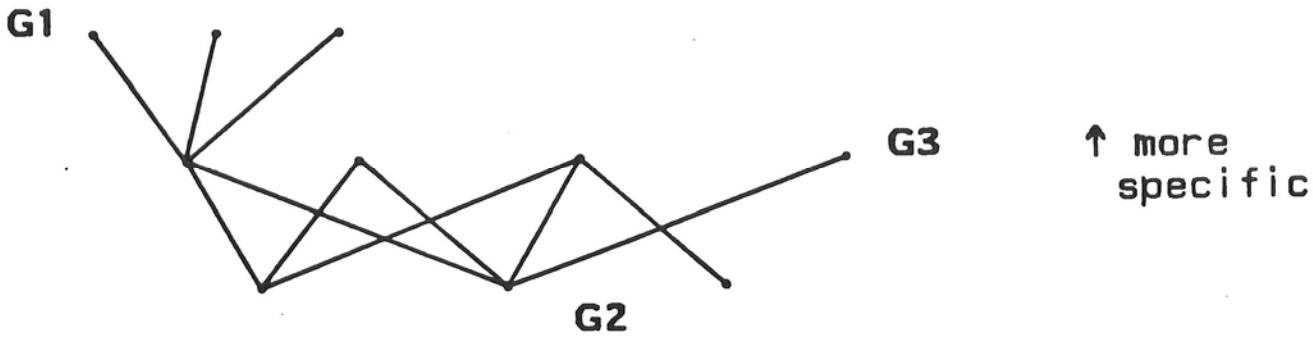
$$\mathbf{G1} \text{ is more-specific-than } \mathbf{G2} \text{ iff } \{ i \in I \mid M(G1,i) \} \subseteq \{ i \in I \mid M(G2,i) \}$$

GENERALIZATION AS SEARCH

G1: { (Large Red Circle) (Large ? ?) }

G2: { (? ? Circle) (Large ? ?) }

G3: { (? ? Circle) (Large Blue ?) }



Strategies

1) Depth-first Search
2) Specific-to-General Breadth-first Search
3) Version Space Bidirectional Search

Concerns:

1) Complexity and Efficiency

2) Using Incompletely Learned Generalizations *
3) Selecting New Training Instances *
4) Prior Knowledge *
5) Inconsistency *

GENERALIZATION AS SEARCH

Depth-first Search: Current Best Hypothesis

For a Negative Instance, i: Find a way to make CBH more specific so that it no longer matches i – and so that it still matches *all* earlier positive instances.

For a Positive Instance, i: Find a way to make CBH more general so that it matches i – and so that it still does not match *any* earlier negative instances.

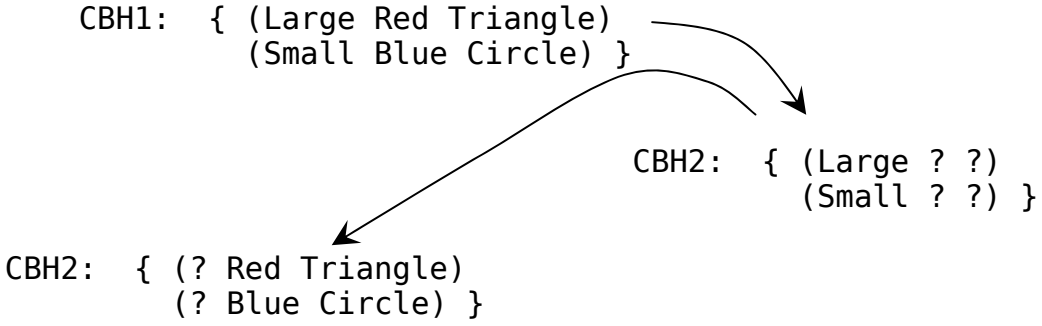
If no acceptable revision to CBH can be found, Backtrack to an earlier version of CBH, try an alternate branch in the search, and reprocess instances seen since then.

Example:

Instances:

```
1. { (Large Red Triangle), (Small Blue Circle) }  +
2. { (Large Blue Circle), (Small Red Triangle) }  +
3. { (Large Blue Triangle), (Small Blue Triangle) }  -
```

Search:



GENERALIZATION AS SEARCH

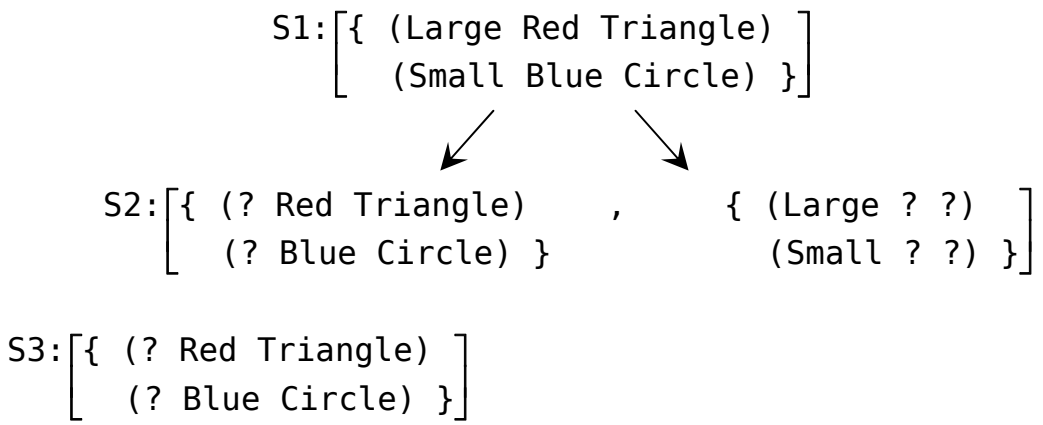
Specific-to-General Breadth-first Search:

Initialize S to the set of *maximally specific* generalizations consistent with the first positive instance.

For a Negative Instance, i: Retain in S only those generalizations that do not match i.

For a Positive Instance, i: Generalize members of S that do not match i along each branch of the partial ordering, just enough so that they match i. Remove from S any member that is either more general than some other member, or that now matches a previous negative instance.

Example:



GENERALIZATION AS SEARCH

Version Space Bidirectional Search:

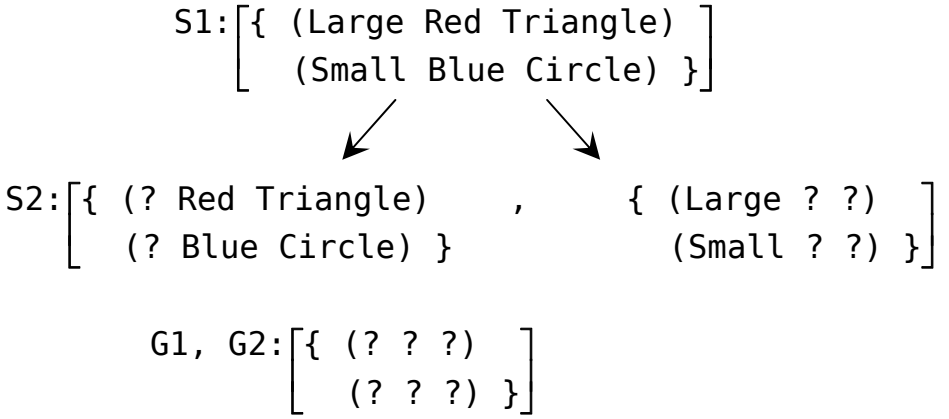
Initialize S to the set of *maximally specific* generalizations consistent with the first positive instance.

Initialize G to the set of *maximally general* generalizations consistent with the first positive instance.

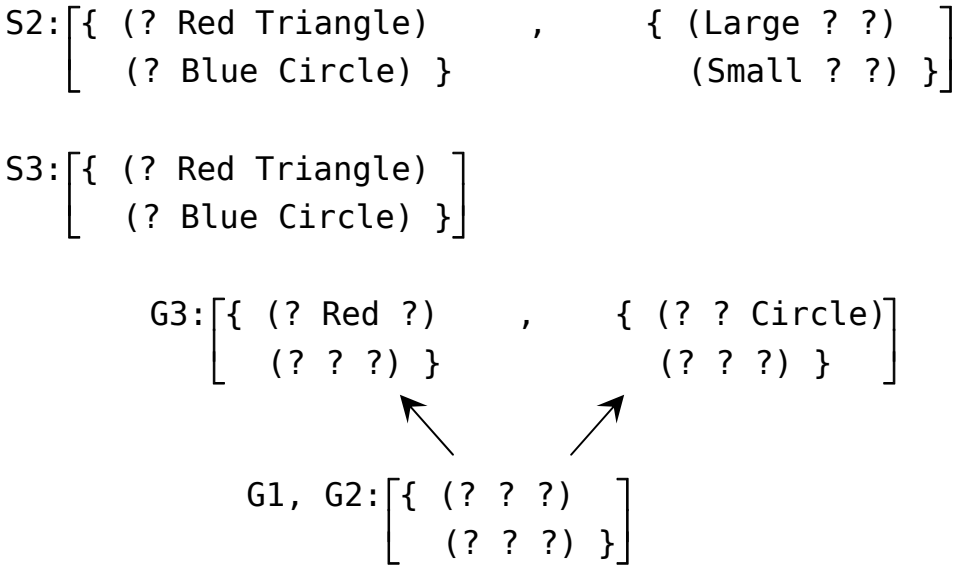
For a Negative Instance, i: Retain in S only those generalizations that do not match i. Make members of G more specific along branches of the partial ordering, in ways such that each member remains more general than some member of S, and just enough so that they no longer match i. Remove from G any member that is more specific than some other member.

For a Positive Instance, i: Retain in G only those generalizations that match i. Generalize members of S along branches of the partial ordering, in ways such that each member remains more specific than some member of G, and just enough so that they match i. Remove from S any member that is more general than some other member.

Example:



GENERALIZATION AS SEARCH



Complexity:

Type	Time	Space
DFS	O(pn)	O(p+n)
BFS	$O(spn+s^2p)$	O(s+n)
VSBS	$O(sg(p+n)+s^2p+g^2n)$	O(s+g)

MACHINE VISION

Can a TV camera see?

Can a computer see?

Assume a computer understands a picture if it can describe the contents in the same terms that people might use, and if it can use the "knowledge" contained in such descriptions for future problem solving.

Low-level problems: resolution, quantization, color, ... a lot of data! (512x512x16x256 ~ 10^9)

Template Matching: Object Identification

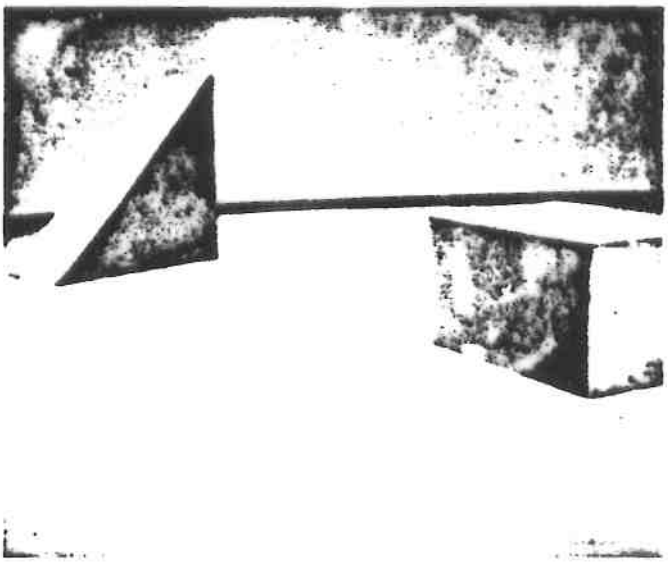
```
0000000000        0000000000        0000000000
0000010000        0000111000        0000111000
0001110000        0001101100        0001000100
0000110000        0000000100        0000001000
0000110000        0000001100        0000011000
0000110000        0000111000        0000001100
0000110000        0001000000        0001000100
0001111000        0001111100        0000111000
0000000000        0000000000        0000000000
0000000000        0000000000        0000000000
```

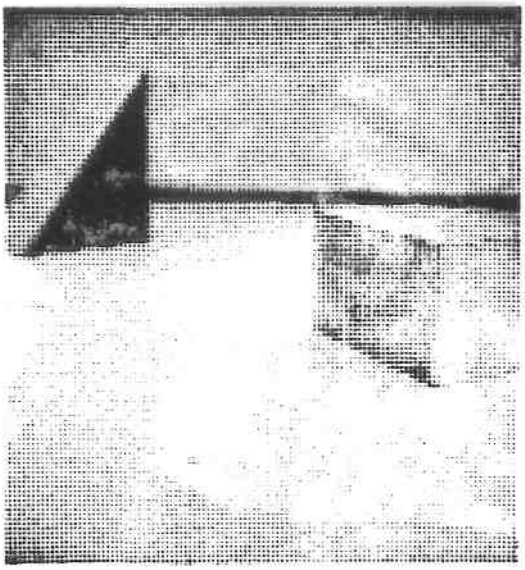
Problems with translation, rotation, scaling. Usually good for <100 patterns.

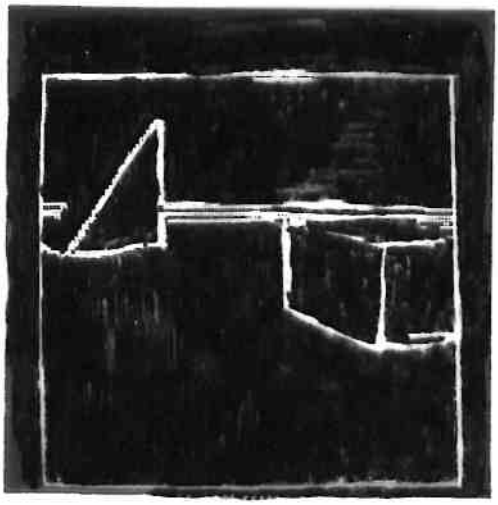
How could we use this scheme to recognize triangles?

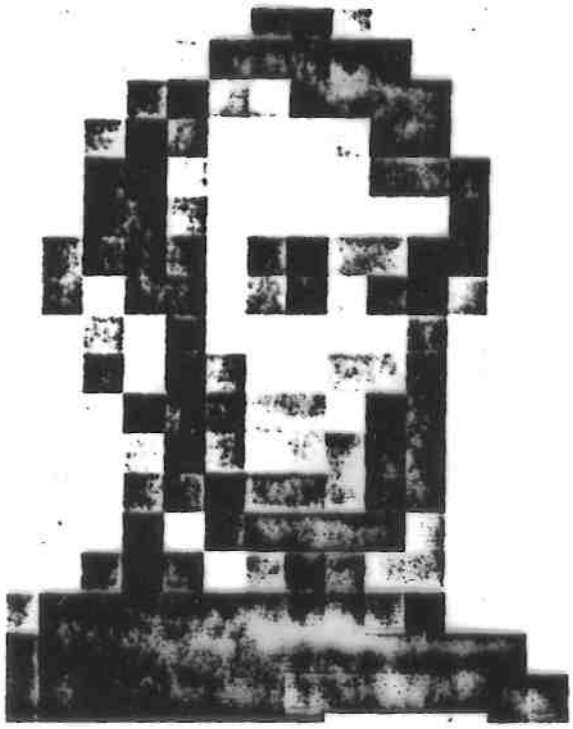
A template is only good for identifying a single shape. A triangle is defined by a description and comes in an infinite variety of shapes.

Proceed via elementary operations, like “find a straight line”, “find a corner”, “follow a contour”, and “count the objects.”







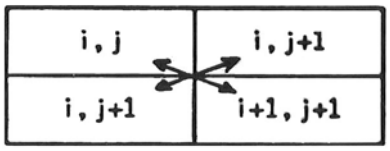


MACHINE VISION

Smoothing: low-pass filtering

Sharpening: spatial differentiation, gradient estimation, edge enhancement.

$$F(i,j) = |g(i,j) - g(i+1,j+1)| + |g(i,j+1) - g(i+1,j)|$$




```
0000000000        0000000000        0000000000
0000000000        0122222100        00+++++000
0011111100        0200000200        0+00000+00
0011111100        0200000200        0+00000+00
0011111100        0200000200        0+00000+00
0011111100        0200000200        0+00000+00
0011111100        0200000200        0+00000+00
0011111100        0122222100        00+++++000
0000000000        0000000000        0000000000
0000000000        0000000000        0000000000
```

Line-Finding:

1) Match local templates. May not be necessary if the sharpening operator has done a good enough job.

−	+	−
−	+	−
−	+	−

−	−	−
+	+	+
−	−	−

−	−	+
−	+	−
+	−	−

+	−	−
−	+	−
−	−	+

2) Form lines.
    1) Add to a group any segment within 3 pixels (picture points) of any segment already in the group and not perpendicular to any segment already in the group.
    2) If a segment cannot be added to an existing group, it begins a new group of its own.
    3) When all such groups have been formed, draw a straight line between the two segments in each group that are farthest apart.

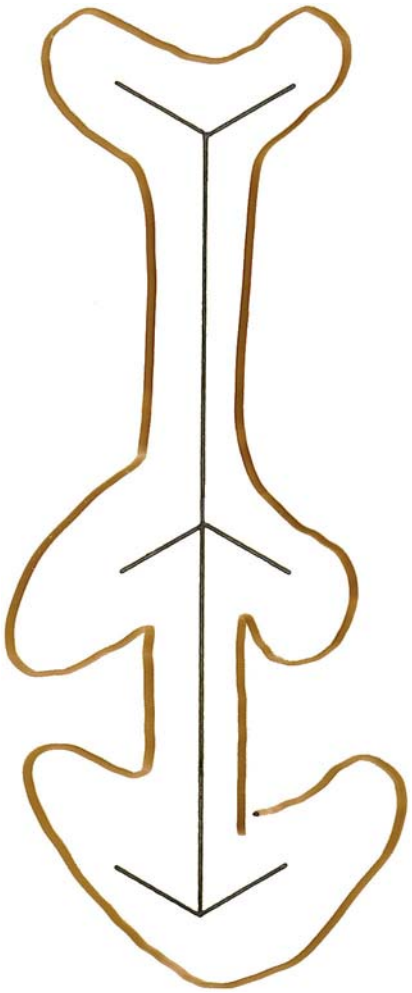
Note ambiguity at corners

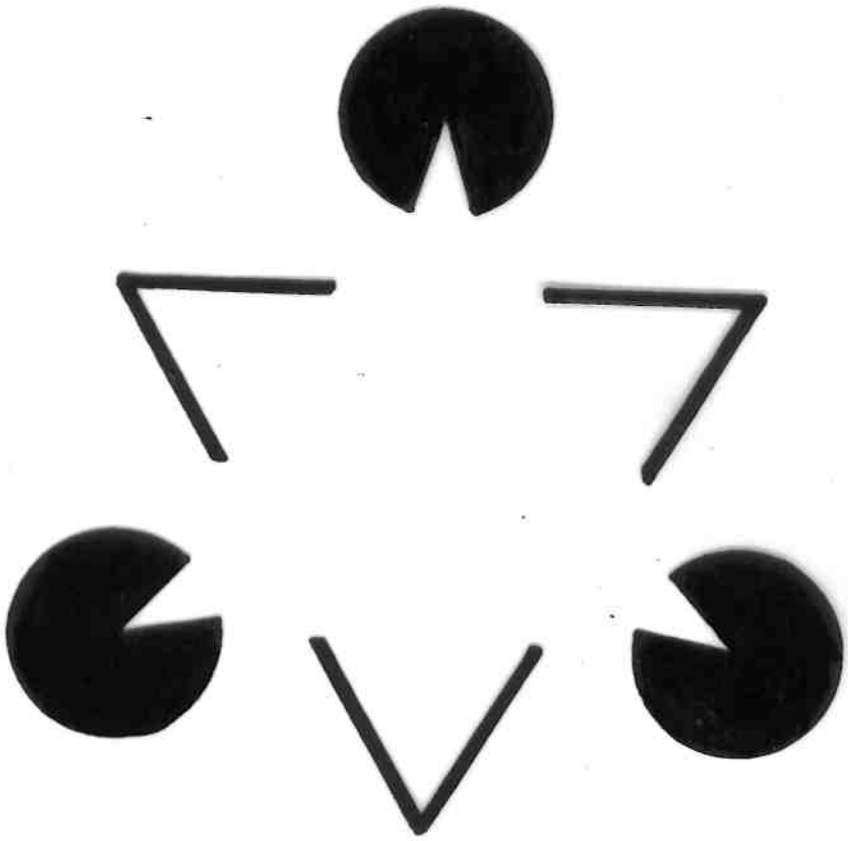
    4) Clean up based on knowledge about objects.

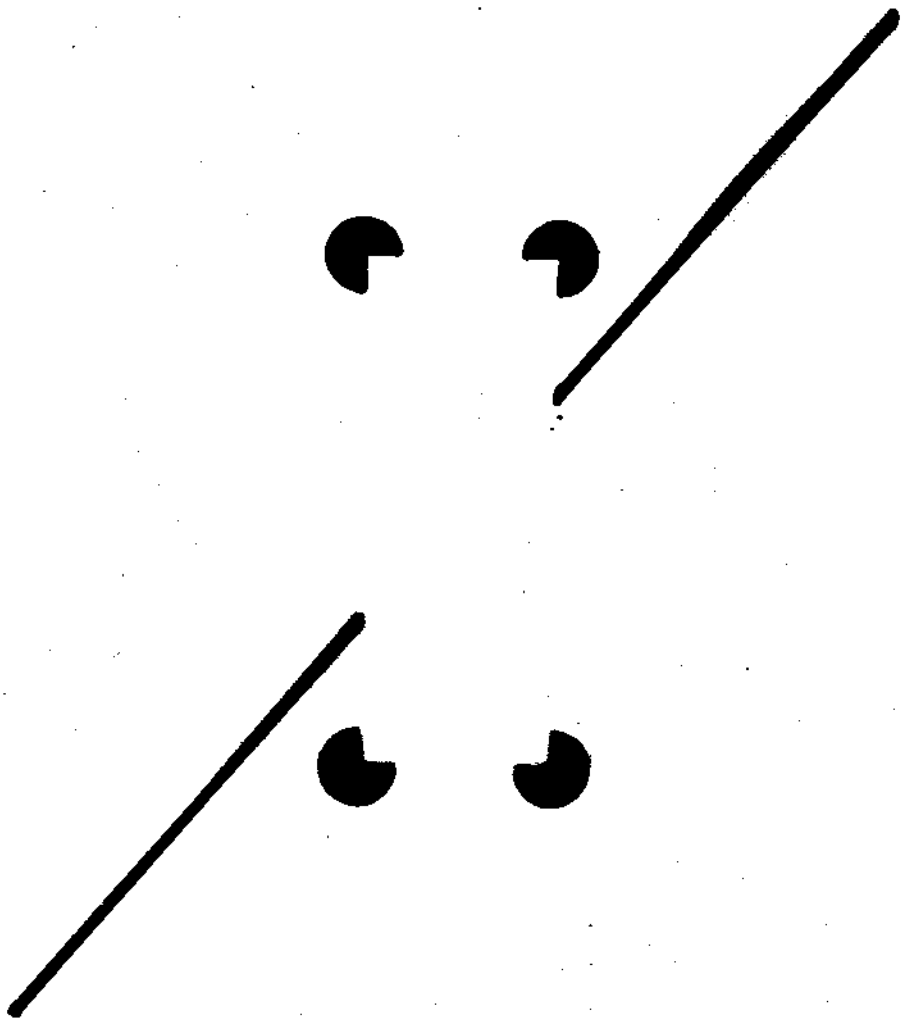
Region Analysis: (Connectivity Analysis)

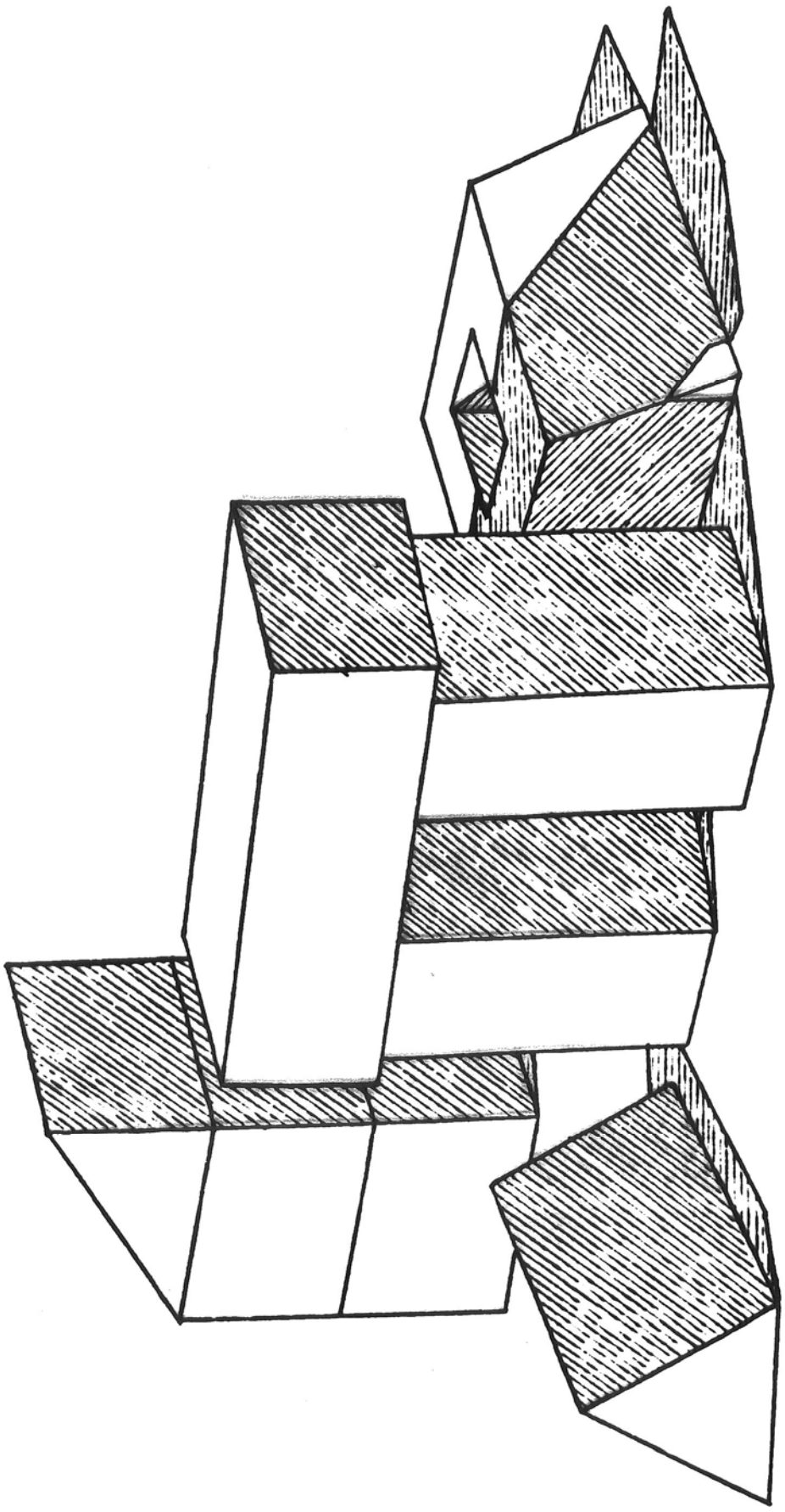
1) Form regions (blobs) on the basis of similar average intensity, connectedness.

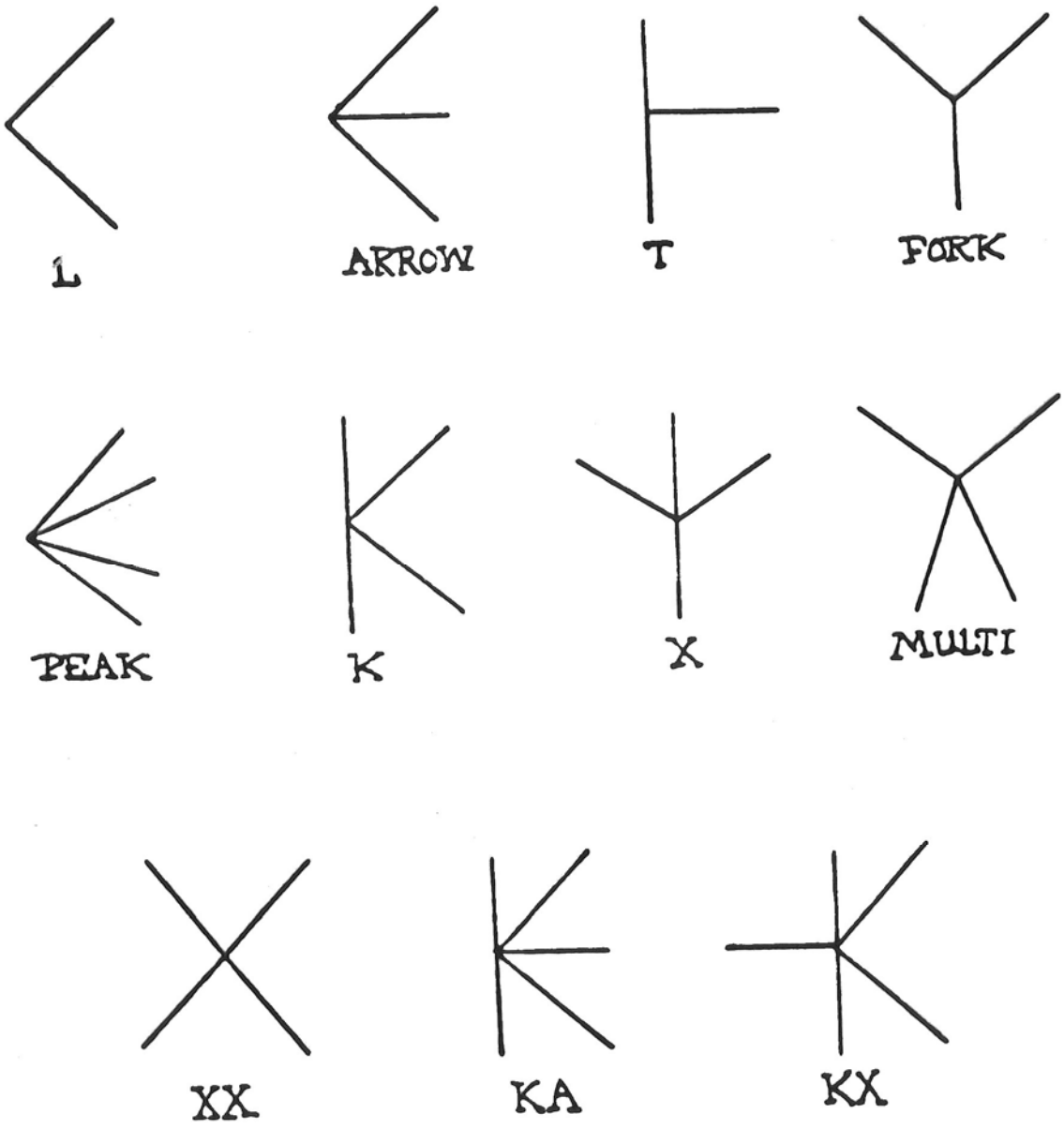
2) Collect feature information about regions:
    1) Maximum limits of extent.
    2) Pointers to surrounded regions (holes) and the surrounding region.
    3) Area.
    4) Centroid.
    5) Axis of least moment of inertia
    6) Perimeter length.
    7) Perimeter point coordinates.

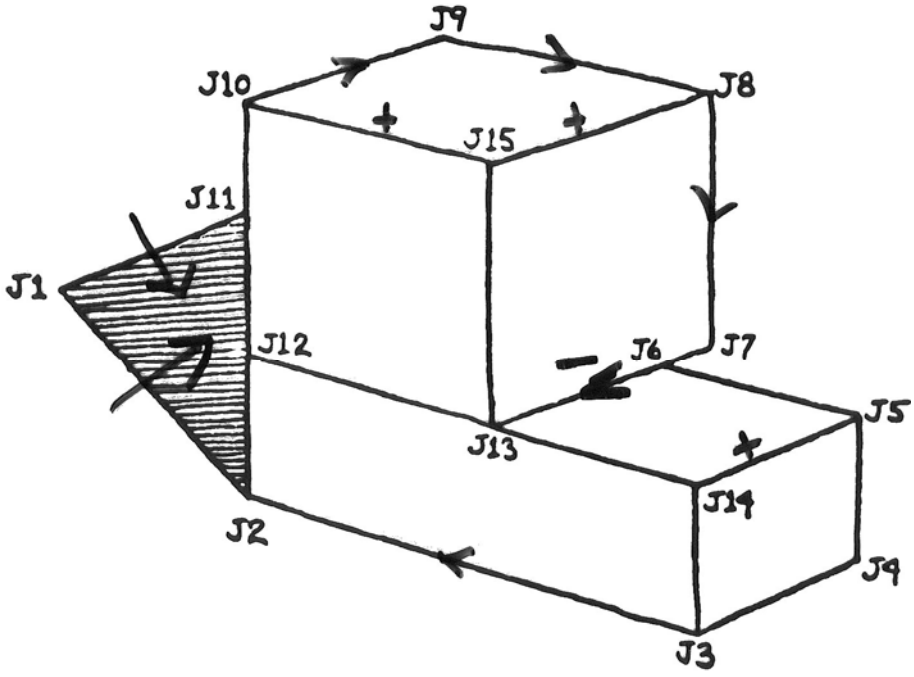








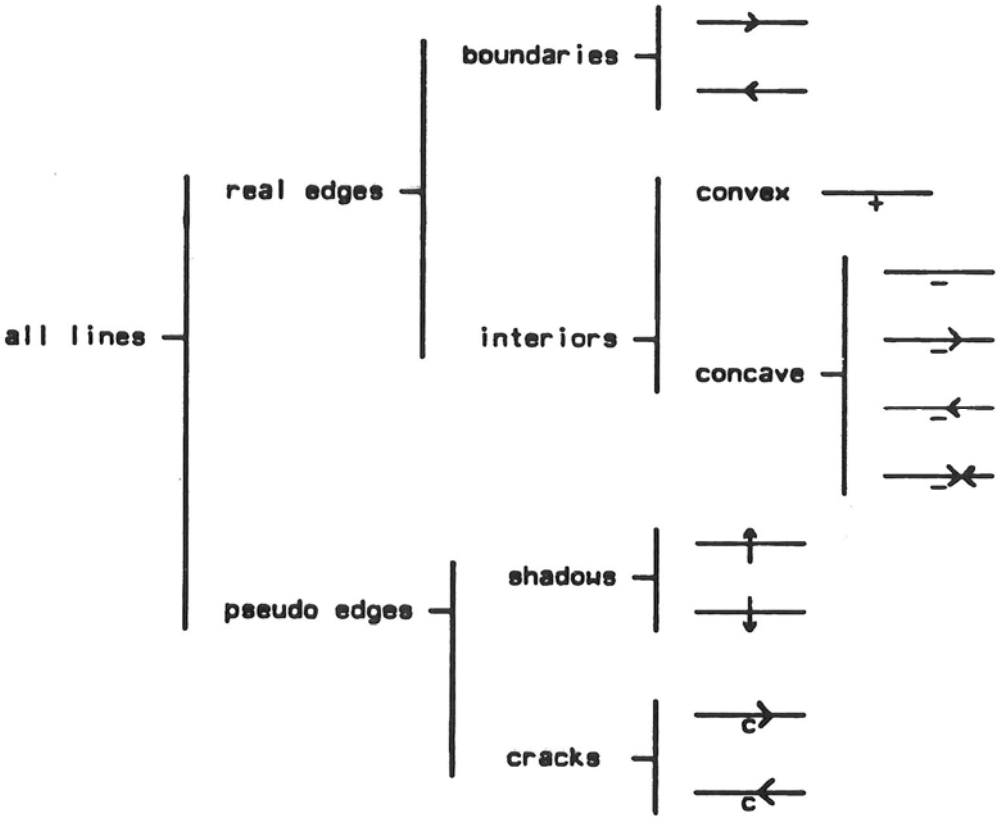

L
ARROW
T
FORK
PEAK
K
X
MULTI
XX
KA
KX



(L)	(ARROW)	(T)	(FORK)	(K)
J1	J2	J6	J14	J13
J4	J3	J11	J15	
J7	J5	J12		
J9	J8			
	J10			

CONSTRAINTS

Huffman/Clowes/Waltz: Classify edges as well as junctions (11 types)



Boundaries and Cracks: Obscuring body lies to *right* of direction of arrow.

Shadows: Arrow points to shadowed region.

Separable Concave Edges: Obscuring body lies to right of direction of arrow; double arrow indicates that three bodies meet along line.

Waltz: Classify regions according to illumination

I - directly illuminated
SP - projected shadow
SS - self-shadowed

INTERPRETATION:

R1 / R2 — AN INSEPARABLE CONCAVE EDGE; THE OBJECT OF WHICH R1 IS A PART [OB(R1)] IS THE SAME AS [OB(R2)].

R1 / R2 — A SEPARABLE TWO-OBJECT CONCAVE EDGE; IF [OB(R1)] IS ABOVE [OB(R2)] THEN [OB(R2)] SUPPORTS [OB(R1)].

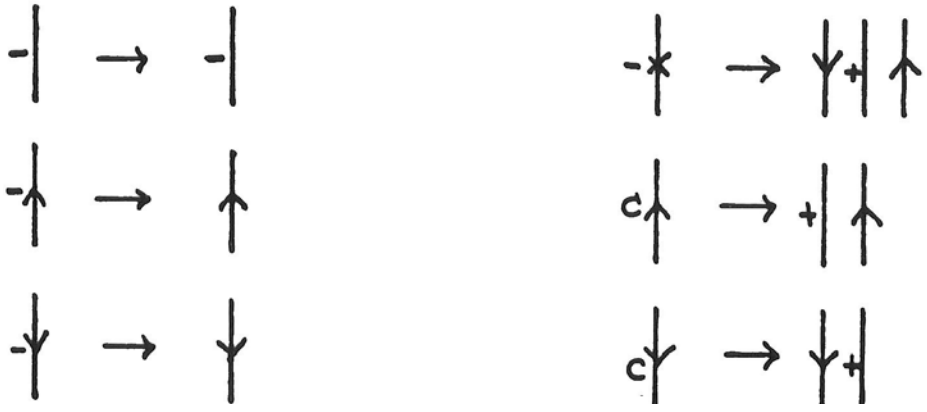
R1 / R2 — SAME AS ABOVE; IF R1 IS ABOVE R2, THEN [OB(R2)] OBSCURES [OB(R1)] OR [OB(R1)] SUPPORTS [OB(R2)].

R1 / R2 — A SEPARABLE THREE-OBJECT CONCAVE EDGE; NEITHER [OB(R1)] NOR [OB(R2)] CAN SUPPORT THE OTHER.

R1 / R2 — A CRACK EDGE; [OB(R2)] IS IN FRONT OF [OB(R1)] IF R1 IS ABOVE R2.

R1 / R2 — A CRACK EDGE; [OB(R2)] SUPPORTS [OB(R1)] IF R1 IS ABOVE R2.

SEPARATIONS:



JUNCTION TYPE	# OF POSSIBILITIES EACH BRANCH LABELED AS INDEPENDENT	ACTUAL # OF NEW JUNCTIONS	
L	3249	92	24
ARROW	185,193	86	
FORK	185,193	826	116
T	185,193	623	
PEAK	11,555,901	10	
X	11,555,901	435	
XX	11,555,901	128	
MULTI	11,555,901	160	
K	11,555,901	213	
KA	6.58×10^8	20	
KX	6.58×10^8	76	

CONSTRAINTS

Relaxation Algorithm: (Constraint Propagation, Range Restriction)

```
FOR each junction DO
  BEGIN
    Attach all possible labels to the junction
    Remove any labels that are impossible given
      the current context of the labels attached
      to neighboring junctions
    Iteratively remove labels from the context
      by propagating the constraints embodied
      in the list of labels for the junction
      outward to the context until no more
      changes can be made in the context.
  END
```

Waltz found computation time proportional to N (number of junctions) instead of M*N (for M labels).

Started with the scene/background boundary [more constraint]

Some junctions do not propagate effects [T junctions]

More knowledge → Less search

[Knowledge *is* Power]