Knowledge-based systems in Bioinformatics, 1MB602, 2007

or

Artificial Intelligence

for Bioinformatics

Torgeir R. Hvidsten

(evolution from slides by Robin Andersson)

What is AI?

"...making a machine behave in ways that would be called intelligent if a human were so behaving"

- John McCarthy, August 31, 1955

"The subfield of computer science concerned with the concepts and methods of symbolic inference by computer and symbolic knowledge representation for use in making inferences."

- The Free On-line Dictionary of Computing (27 SEP 03)

Definitions of AI

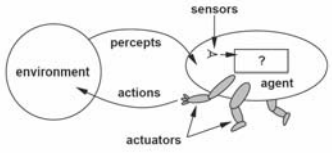
Four categories of definitions:

	Empirical science	Engineering
	Human-centered	**Rationality-centered**
Reasoning	Systems that think like humans	Systems that think rationally
Behavior	Systems that act like humans	Systems that act rationally

Acting humanly: Turing test

- Turing proposed that a computer program show intelligent behavior if is able to fool a human interrogator:
- The Turing test: the computer is interrogated by a human via a teletype, and passes the test if the interrogator cannot tell if there is a computer or a human at the other end
  - natural language processing
  - knowledge representation
  - automated reasoning
  - machine learning
  - (computer vision and robotics)

Intelligent agents



- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators
- An agents behavior can be described as an agent function that maps any percept sequence to an action

AI techniques

- Logics
- Knowledge engineering
- Search
- Machine learning
- Pattern recognition
- Automatic theorem proving
- Planning
- Machine vision
- Natural language processing

Knowledge-based systems in Bioinformatics

Boolean Methods in Bioinformatics

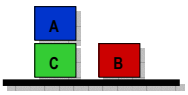
Logics for AI

One needs a logical languages for describing states and actions

A logic consists of:

1. A formal language for describing states of affairs:
   a) the syntax of the language describes how to make sentences
   b) the semantics of the language states what a sentence means, in particular, if it is true or false
2. A proof theory – a set of rules for inferring new sentences (theorems) from a set of existing sentences

Logic for AI



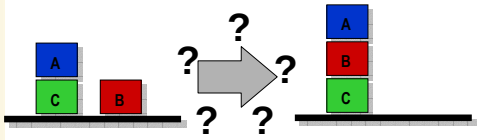

```
on(a,c)        clear(a)
ontable(c)     clear(b)
ontable(b)
```

```
∀X,Y(((clear(X) ∧ clear(Y) ∧ ontable(X)) ∨
∃Z(clear(X) ∧ clear(Y) ∧ on(X,Z)))
⇒ valid_move(X,Y))
```

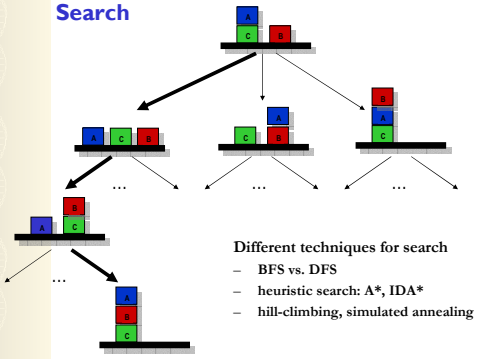

Planning

Generate a strategy for achieving goals (sequence of actions) from:

- facts about the world and the particular situation
- statement of a goal



Search



Different techniques for search

- BFS vs. DFS
- heuristic search: A*, IDA*
- hill-climbing, simulated annealing

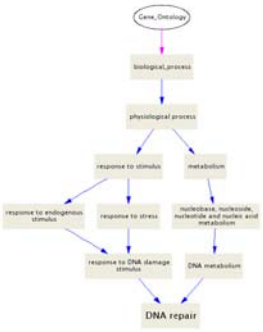
Examples of search problems

- Chess: search through the set of possible moves
  - Looking for one which will best improve position
- Route planning: search through the set of paths
  - Looking for one which will minimal distance
- Theorem proving:
  - Search through sets of reasoning steps looking for a reasoning progression which proves a theorem
- Machine learning:
  - Search through a set of concepts looking for a concept which achieves target categorisation

Knowledge representation

- Logical representations
  - Defined language
  - Enables "logical reasoning"
- Frames
  - Information retrieval in face of new situations
- Ontology organization
  - "specification of conceptualizations": description of concepts and relations that exists in a specific domain

Knowledge representation cont.



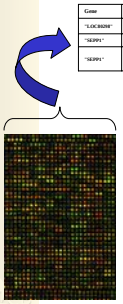
goterm(GO:0006281, "DNA repair").
isa(GO:0006281, GO:0006259).
isa(GO:0006281, GO:0006974).
goterm(GO:0006974, "response to DNA damage stimulus").
isa(GO:0006974, GO:0006950).
isa(GO:0006974, GO:0009719).
goterm(GO:0006259, "DNA metabolism").
isa(GO:0006259, GO:0006139).
goterm(GO:0006950, "response to stress").
isa(GO:0006950, GO:0050896).
goterm(GO:0009719, "response to endogenous stimulus").
isa(GO:0009719, GO:0050896).
goterm(GO:0006139, "nucleobase, nucleoside, nucleotide and nucleic acid metabolism").
isa(GO:0006139, GO:0008152).
goterm(GO:0050896, "response to stimulus").
isa(GO:0050896, GO:0007582).
goterm(GO:0008152, "metabolism").
isa(GO:0008152, GO:0007582).
goterm(GO:0007582, "physiological process").
isa(GO:0007582, GO:0008150).
goterm(GO:0008150, "biological_process").
isa(GO:0008150, top).

Machine learning

- Supervised learning
  - Learn from training examples, predict/classify test examples
    - Neural networks
    - Bayesian statistics
    - Decision tree learning
    - Rough set theory
- Unsupervised learning
  - Models are built without a well defined goal or prediction outcome
    - Clustering techniques

Supervised learning

Predicting biological function of genes from microarray expression profiles



Gene	0MIN	15MIN	30MIN	1H	2H	4H	6H	8H	12H	16H	20H	24H	Annotation
"LOC8288"	0.00	-0.24	-1.90	-2.07	-0.35	-0.25	-0.98	-1.20	-1.44	-1.42	-0.8	-0.94	unknown
"SEPP1"	0.00	0.33	0.37	0.14	0.25	-0.3	-0.67	-0.37	-0.29	-0.22	-0.3	-0.42	"transport"
"SEPP1"	0.00	0.33	0.37	0.14	0.25	-0.3	-0.67	-0.37	-0.29	-0.22	-0.3	-0.42	"stress response"

```
IF 0h-4h(Increasing) AND 6h-10h(Decreasing) AND
14h-18h(Constant)
THEN Annotation(cell proliferation) OR
Annotation(cell-cell signalling) OR
Annotation(intracellular signalling cascade)
```

Predict unknown gene function by applying decision rules from genes with known function

What this course is about

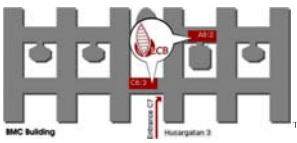
- Goal: Introduction to intelligent bioinformatics
  - Logic and knowledge representation
  - Heuristic search
  - Logical inference
  - Probabilistic approaches
  - Genetic algorithms and genetic programming
  - Decision trees and neural networks
  - Cellular automata
- Programming languages for AI
  - Functional programming
    - Scheme (LISP)

Why yet another programming language?

- Different languages have different strengths and weaknesses
  - Imperative programming: describes computation as statements that change a program state (e.g. Fortran, C, and Java)
  - Functional programming: treats computation as the evaluation of (mathematical) functions, and avoids state (e.g. LISP)
  - Declarative versus imperative programming: imperative programs explicitly specify an algorithm to achieve a goal, while declarative programs explicitly specify the goal and leave the implementation of the algorithm to the support software
- The best way to learn programming is to acquire skills in different languages
- Learning a new language is fun!

Practical information

- http://www.lcb.uu.se/~hvidsten/KS
- Examination
  - Exercises: 1p
  - Project work: 1p
  - Written exam: 3p
- Lecturer
  - Torgeir R. Hvidsten, hvidsten@lcb.uu.se
    office: A6:313d, phone: 471 6687



Knowledge-based systems in Bioinformatics, 1MB602

Lecture 1:
Scheme basics

Lecture overview

- Part 1: The Scheme language
  - Language elements
  - Evaluation
  - lambda and define
- Part 2: Procedures and processes
  - Substitution model
  - Recursion
  - Types
  - Iteration

Scheme

1. Functional programming language
2. (Almost) every expression has a value, which is returned when the expression is evaluated
3. Every value has a type

Language elements

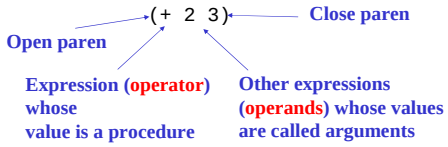
- Primitives
- Means of combination
- Means of abstraction

Primitives

- Numbers – self-evaluating
  `23` → `23`
  `-36` → `-36`
- Names for built-in procedures
  `+`, `*`, `/`, `-`, `=`, …
  - What is the value of such an expression?
    `+` → `#<procedure>`
  - Evaluated by looking up the value associated with the name

Combinations

How do we create expressions using procedures?



Combinations are evaluate by getting the values of its subexpressions, and then applying the operator (procedure) to the operands (arguments)

Combinations cont.

Nested combinations are evaluated recursively:

```
(+ (* 2 3) 4)          → 10

(* (+ 3 4) (- 8 2))    → 42

(* (+ 5 (/ 2 3))
   (+ (* 4 6) -1))     → 391/3
```


Abstraction

- In order to abstract an expression, we need a way to give it a name

```
(define score 23)
(define total (+ 12 13))
```

- `define` is a special form
  - Does not evaluate the second expression
  - Rather, it pairs that name with the value of the third expression in an environment
  - Return value is unspecified
- `(* 100 (/ score total))` → `92`

Rules for evaluation

1. If self-evaluating, return value
2. If a name, return the value associated with that name in the environment
3. If a special form, do something special
4. If a combination, then
   1. Evaluate all of the subexpressions of the combination (in any order)
   2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (operands), and return the result

Abstraction cont.

- How do we generalize the following expression?

```
(* x x)
```

- Need to capture ways of doing things: use procedures

```
(lambda (x) (* x x))
```

(formal) parameters

body

To process something multiply it by itself

- `lambda` is a special form – creates a procedure and return it as a value

Abstraction cont.

- Use this anywhere you would use a procedure

```
((lambda (x) (* x x)) 5)
(* 5 5)
25
```

- Can give it a name

```
(define square (lambda (x) (* x x)))

(square 5) → 25
```


Rules for evaluation of procedures

1. If the procedure is a primitive procedure, just apply it
2. If the procedure is a compound procedure, then evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value

Mathematical operators are just names

```
1. (+ 3 5)              →   8
2. (define fred +)      →   #<procedure>
3. (fred 4 6)           →   10
```

How to explain this?

- `+` is just a name
- `+` is bound to a value which is a procedure
- Line 2 binds the name `fred` to that same value (which is a procedure)

Interaction of define and lambda

```
1. (lambda (x) (* x x))
              → #<procedure>

2. (define square (lambda (x) (* x x)))
              → square

3. (square 4)                       → 16

4. ((lambda (x) (* x x)) 4)         → 16

5. (define (square x) (* x x))      → square
```

- a convenient shorthand for 2 above
- this is a use of lambda!

The value of a lambda expression is a procedure!

Your turn: fill in each box

- `(define twice (lambda (x) (* 2 x)))`
  `(twice 2)` → `4`
  `(twice 3)` → `6`

- `(define constant2 (lambda () 2))`
  `(constant2)` → `2`

- `(define second (lambda (x y z) y))`
  `(second 2 15 3)` → `15`
  `(second 34 -5 16)` → `-5`

End of part 1: review

- Things that make up Scheme programs:
  - Self-evaluating: `23`, `"hello"`, `#t`
  - Names: `+`, `pi`
  - Combinations: `(+ 2 3)` `(* pi 4)`
  - Special forms: `(define pi 3.14)`
- Syntax
  - Combination: (operator-expr operand-exprs …)
  - Special form: the left-most sub expression is a special keyword
- Semantics
  - Combinations: evaluate sub expressions in any order, apply operator to operands
  - Special forms: each one special

Substitution model

- A method that explains what happens during evaluation
- To apply a compound procedure:
  - Evaluate the body of the procedure, with each parameter replaced by the corresponding arguments
- To evaluate a primitive procedure: just apply it

- ```
  (define (square x) (* x x))
  (square 4)
  (* 4 4)
  16
  ```


Substitution model details

- ```
  (define (square x) (* x x))
  ```
- ```
  (define (average x y)
      (/ (+ x y) 2))
  ```

```
(average 5 (square 3))
(average 5 (* 3 3))
(average 5 9)
```

first evaluate operands, then substitute (applicative order)

```
(/ (+ 5 9) 2)
(/ 14 2)
7
```

if operator is a primitive procedure, replace by result of operation
```

Applicative vs. normal order

- Applicative order
  - Scheme is an applicative order language
  - All the arguments to Scheme procedures are evaluated before the procedure is applied
- Normal order (lazy evaluation)
  - Delay evaluation of procedure arguments until the actual argument values are needed
- What is the result of evaluating the following expression in each evaluation principle?

```
(define (try a b)
  (if (= a 0) 1 b))

(try 0 (/ 1 0))
```


The factorial procedure

- Compute n factorial, defined as n! = n(n-1)(n-2)(n-3)…1
- Notice that n! = n*[(n-1)(n-2)…] = n*(n-1)! If n > 1

```
(define (fact n)
   (if (= n 1)
       1
       (* n (fact (- n 1)))))
```

- Predicate = tests numerical equality

```
(= 4 4) → #t (true)
(= 4 5) → #f (false)
```

- if is a special form

```
(if (= 4 4) 2 3) → 2
(if (= 4 5) 2 3) → 3
```



```
(define (fact n)
  (if (= n 1) 1 (* n (fact (- n 1)))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f 1 (* 3 (fact 2)))
(if #f 1 (* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1))))))
(if #f 1 (* 3 (if #f 1 (* 2 (fact (- 2 1))))))
(if #f 1 (* 3 (if #f 1 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1)))))))))
(if #f 1 (* 3 (if #f 1 (* 2 (if #t 1 (* 1 (fact (- 1 1)))))))))
(if #f 1 (* 3 (if #f 1 (* 2 1))))
(if #f 1 (* 3 (if #f 1 2)))
(if #f 1 (* 3 2))
(if #f 1 6)
6
```


Recursive algorithms

- The fact procedure is a recursive algorithm
- A recursive algorithm:
  - In the substitution model, the expression keeps growing

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
```

  - Different types of recursion
    Tree recursion, tail recursion, …
- Iterative algorithms:
  - In the substitution model, the expression size is constant

Design of recursive algorithms

- Decompose the problem
  - Solve a problem by:
    1. Solving a smaller instance (wishful thinking)
    2. Converting that solution into the desired solution
  - Step 2 requires creativity!
    - Must design the strategy before coding
    - n! = n(n-1)(n-2)… = n[(n-1)(n-2)…] = n * (n-1)!
    - Solve the smaller instance, multiply it by n to get the solution

```
(define (fact n)
   (* n (fact (- n 1))))
```


Design of recursive algorithms cont.

- Identify non-decomposable problems
  - Decomposing is not enough by itself
  - Must identify the "smallest" problems and solve them directly
  - Define 1! = 1

```
(define (fact n)
   (if (= n 1) 1
         (* n (fact (- n 1)))))
```

General form of recursive algorithms

- Test, base case, recursive case

```
(define (fact n)
  (if (= n 1)                    ; test for base case
      1                          ; base case
      (* n (fact (- n 1)))       ; recursive case
))
```

- base case: smallest (non-decomposable) problem
- recursive case: larger (decomposable) problem

Iterative algorithms

- In a recursive algorithm, bigger operands => more space

```
                (define (fact n)
                   (if (= n 1) 1
                       (* n (fact (- n 1))))))
   (fact 4)
   (* 4 (fact 3))
   (* 4 (* 3 (fact 2)))
   (* 4 (* 3 (* 2 (fact 1))))
   (* 4 (* 3 (* 2 1)))
   ...
   24
```

Pending operation

- Pending ops makes the expression grow continuously
- An iterative algorithm uses constant space

Iterative factorial algorithm

- Intuition: same as you would do if calculating factorial by hand, e.g 4!:

4 * 3 = 12	or	1 * 2 = 2
12 * 2 = 24		2 * 3 = 6
24 * 1 = 24		6 * 4 = 24
=> 4! = 24		=> 4! = 24

- At each step, we only need to remember:
  - Previous product
  - Next multiplier

Iterative factorial algorithm cont.

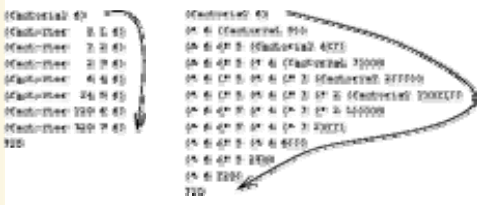
```
(define (ifact-help n product count)
     (if (> count n)
          product
          (ifact-help n
                      (* product count)
                      (+ count 1)))))

(define (ifact n)
   (ifact-help n 1 1))

(ifact 4)
(ifact-help 4 1 1)
(ifact-help 4 1 2)
(ifact-help 4 2 3)
(ifact-help 4 6 4)
(ifact-help 4 24 5)
```

Fixed space size because no pending operations

Iterative versus recursive processes



Types

- `(+ 5 10)` → 15
- `(+ 5 "hi")` →

The object "hi", passed as the second argument to integer-add, is not the correct type.

- Addition is not defined for strings
- The type of the integer-add procedure is

number, number → number

two arguments, both numbers

result value of integer-add is a number

Type examples

expression	evaluates to a value of type
`15`	number
`"hi"`	string
`square`	number → number
`>`	number, number → boolean

- **The type of a procedure is a** contract**:**
  - **If the operands have the specified types, the procedure will result in a value of the specified type**
  - **Otherwise, its behavior is undefined: Maybe an error, maybe random behavior**

References

- **S. Russell, P. Norvig, Artificial intelligence: a modern approach, Prentice-Hall, Upper Saddle River, New Jersey, 1995**
- **H. Abelson, G.J. Sussman, Structure and Interpretation of Computer Programs 2nd ed, The MIT Press, Cambridge, Massachusetts, 2000, Chp: 1.0-1.2, pp: 1-53**
- **6.001 Spring 2000: Lecture Notes, lecture 1-2,4, http://sicp.ai.mit.edu/Spring-2000/lectures/**

LCB
THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se