Computer science

Adam Boult (www.bou.lt)

January 22, 2021

Contents

Preface **2**

I Foundations **3**

1 Combinational logic **4**

2 Representing and using integers and real numbers **7**

3 Finite-state machines, pushdown automatons and Turing machines **11**

4 Machine code **13**

II Low-level programming **14**

5 Assembly and high-level programming lanaguges **15**

III Data structures **20**

6 Lists **21**

7 Characters and strings **22**

8 Objects **23**

9 Databases **26**

10 Queues **28**

11 Pointers **29**

IV High-level programming 30

12 Imperative and functional programming 31

13 Object-Oriented Programming 33

14 Emulation 34

V Algorithms 35

15 Algorithms 36

16 Algorithmic complexity 39

VI Algorithmic paradigms 40

17 Dynamic programming 41

18 Parallel processing 43

19 Divide and conquer algorithms 45

20 Greedy algorithms 46

VII Elementary algorithms 47

21 Sorting and searching lists 48

22 Tree search 50

23 Tree search with heuristics 53

24 Constraint Satisfaction Problem (CSP) 55

VIII Numerical methods 58

25 Floating point maths 59

26 Discrete maths 60

27 Linear algebra 62

28 Lossless compression 64

IX Optimisation **65**

29 Unconstrained optimisation **66**

30 Constrained optimisation **71**

X Improving knowledge **72**

31 Knowledge bases **73**

32 Expert systems **76**

XI Applications **77**

33 Image manipulation **78**

34 3D modelling **79**

Preface

This is a live document, and is full of gaps, mistakes, typos etc.

Part I

Foundations

Chapter 1

Combinational logic

1.1 Registers

1.1.1 The bit

A single bit can store a binary piece of information. We can use it to distinguish between two states.

These two states could be represented by True T and False F, but by convention we use 1 and 0.

We can combine bits to store more complex pieces of information. If we have n bits, we can distinguish between 2^n states.

Eight bits together constitute a byte. This can represent one of $2^8 = 256$ states.

1.1.2 Registers

1.1.3 Representing finite states

1.2 Editing the register

1.2.1 Setting register values

1.3 Unary bitwise operations

1.3.1 NOT

We can perform basic operations on inputs.

A simple operation is the unary NOT operator, which returns the reverse of the values of all bits.

We also have binary operators, which take two bits and return another bit. Of note are, AND, OR and XOR.

These operations are a model of Boolean algebra. So there are 16 possible binary functions and 4 possible unary operators.

As in logic, we can combine elementary logical gates to create other logic gates.

These operations can also be performed on a series of bits, however each individual bit is independent of other bits for these operations.

1.3.2 Bitshifts

We can have other operations where bits do interact. An important operator is the bit shift. This takes a series of bits and shifts them to the right or left by one place. This pushes one bit off the end.

The new bit can take a 0 or 1.

Logic gates are not needed for bit shifts. Instead wiring of inputs to outputs achieves the same effect.

1.4 Bit arrays

1.4.1 Registry addresses

1.4.2 Opcodes

1.4.3 Bit arrays, bytes and nibbles

1.4.4 Words

1.4.5 Bit fields

1.5 Binary bitwise operations

1.5.1 AND, OR and XOR

1.5.2 Combining operations

Simple operators, such as bitwise operators and bit shifts, can be combined to create more complex operators. These could have a large number of inputs, outputs and logical gates.

1.5.3 Cycles per instruction

1.5.4 Instruction pipelines

1.5.5 Stack

1.5.6 Stack overflow

Chapter 2

Representing and using integers and real numbers

2.1 Natural numbers

2.1.1 Representing natural numbers

We will use a byte to describe natural numbers. This gives us a range of $2^8 = 256$. As we include 0 the largest number here is 255.

2.1.2 Zero

We describe zero using all 0s.

$0 = 00000000$

2.1.3 Natural numbers

We show other natural numbers by iterating through the possible bits. To increase by one we take the last bit and use the NOT operator on it. If this bit is 1 we are done.

If this bit is now 0 we move to the next bit and flip that. We then repeat the check and repeat until we either run out of bits, or flip a bit to 1.

2.1.4 Arithmetic Logic Units (ALUs)

Half adder

We first introduce the half adder

Take two inputs A and B and put both inputs to two binary operators.

The operators are an XOR gate and an AND gate.

The AND gate only returns 1 if both inputs are 1. The XOR gate returns 1 if only one input is 1.

The XOR gate returns the second digit while the AND date returns the first.

Full adder

This can be used to add 0 and 1, but not larger numbers. To do this we introduce the full adder. This adds two numbers, as before, and allows any remainder from the previous addition to be carried forward.

Full adders can be stringed together to add larger numbers.

Subtraction

Subtraction works similarly to addition. The equivalent half adder returns different values, and the carry forward is not used for addition, but for subtraction.

2.1.5 Multiplication and division of natural numbers

mult x2 is bitshiftleft, /2 is bitshift right) (how to divide when not divisible)

2.1.6 Mod and remainders

2.1.7 Powers, logarithms and exponentials

2.2 Integers

2.2.1 Representing integers

We can expand the natural numbers to the integers.

Consider a byte representing the natural numbers. Previously this would have gone from 0 to 255, with a series of all 1s representing 255.

To introduce integers all numbers with a 1 in the leftmost bit are considered to be negative.

2.2.2 Two's complement

We represent the value of these negative numbers with twos complement. With twos complement the number after 127 is -128. Note that this does not just use the first bit as a sign. The use of twos complement allows us to use the arithmetical logical units for the integers.

2.2.3 Using ALUs with integers

We can expand the natural numbers to the integers.

Consider a byte representing the natural numbers. Previously this would have gone from 0 to 255, with a series of all 1s representing 255.

To introduce integers all numbers with a 1 in the leftmost bit are considered to be negative.

2.2.4 Two's complement

We represent the value of these negative numbers with twos complement. With twos complement the number after 127 is -128. Note that this does not just use the first bit as a sign. The use of twos complement allows us to use the arithmetical logical units for the integers.

2.3 Real numbers

2.3.1 Representing real numbers

store as two integers (x, y) evaluate as $x * 2^y$ this is binary floating point this means you get inaccuracies eg $(0.1 + 0.2 - 0.3) * 10^20$ is not zero alternative is decimal floating point store as $x * 10^y$

2.3.2 Addition, sub, mult, div of real

2.3.3 Powers, log, exp

2.3.4 Floor and ceiling

2.3.5 Floating-Point Units (FPUs)

2.3.6 Overflow and underflow

The need to approximate real operations with pseudo real numbers. If we round small values to 0, then ln 0, x/0 break. This is underflow

2.3.7 Overflow

2.3.8 Underflow

Chapter 3

Finite-state machines, pushdown automatons and Turing machines

3.1 Finite-state machines

3.1.1 Finite-state machines

A finite-state machine has n states. The machine moves through these states through the use of inputs, and the movement between these states can be represented by a graph.

We can represent the machine with table. One axis shows the inputs, the other the states. The cells can be filled with (if available) movements between the states.

Similar to combinatorial logic, but output is the new state.

3.2 Pushdown automatons

3.2.1 Pushdown automatons

In a finite-state machine the action of the machine depends on the state, and on the input. In a pushdown automaton it also depends on a third input stack.

The stack is a list of symbols. It is the first item on the stack which is used to inform the decision.

The actions of a machine in finite-state machine were to move from state to state. In a pushdown automaton, the actions can also affect the stack, and therefore future changes in state.

A pushdown automaton uses a Last-in First-out (LiFo) stack.

3.3 Turing machines

3.3.1 Turing machines

A Turing machine is a pushdown automaton where the stack is no longer LiFo.

3.3.2 Church-Turing thesis

Chapter 4

Machine code

4.1 Programs

4.2 Control flow

4.2.1 Branch

4.2.2 Indirect branch

4.2.3 Conditional branch

4.2.4 Calls

4.2.5 Branch prediction

Will a conditional jump be taken?

4.2.6 Branch target prediction

Predict where to go next

Part II

Low-level programming

Chapter 5

Assembly and high-level programming lanaguges

5.1 Assembly code

5.1.1 Assembly language

Rather than machine code, programs can be written in human readable form. For example, we can use short English language strings instead of hexidecimal or binary.

5.1.2 Assemblers

Assembly code can be converted to machine code using an assembler.

The assembler takes the assembly code as input and returns machine code.

5.1.3 Stack buffer overflow

5.2 High-level programming languages

5.2.1 Compilers

Directly to machine code

5.2.2 Interpreters

Evaluate each line as it is run

5.2.3 Trans-compilers

Compile to code in intermediate language which has its own compiler for further compiling

5.2.4 Portability

5.2.5 Debugging

5.3 Variables

5.3.1 Assigning variables

5.3.2 Direct storage and pointers

Variables can either refer to a part of memory, reserving that space for just that variable, alternatively, rather than store value, we can store pointer.

5.3.3 Assigning variables from other variables

5.3.4 Static and dynamic typing

5.3.5 Mutable and immutable variables

5.3.6 Garbage collection

5.3.7 Pointers

So in first case we have $x = 2$ and $y = 2$, and then we want to change both to 3.

In first example, initiate both and store 2 in memory twice. update both.

In second example we can have x and y have the same value of a pointer, pointing to a third memory location. we can the update this memory location to change both to 3.

Two variable can be the same by value, but additionally by pointer.

eg $x = 2$ and $y = 2$. $x == y$, but if they have the same pointer, they are the same thing. changing one changes the other.

We can update x and y will automatically update

When we say $x = 2$, $y = x$ we are either making y mutable or immutable. mutable means same pointer. language specific rules apply.

5.3.8 Integer caching

If we set $x = 2$ we can either create 2 in memory, or simply point x to 2, which is already in memory

That means if we do $x = 2$ $y = 2$ they have the same pointer.

Can also cache some other common data values, eg empty lists.

Makes sense if pointer is smaller in memory than value.

5.4 Functions

5.4.1 Macros

Not separate, code is replaced before it is run

5.4.2 Subroutines

5.4.3 Libraries

5.4.4 Global and local variables

5.4.5 Call by value and call by name

Two ways of giving args to parameters, value evals when func called, name evals whenever needed.

5.4.6 Memory allocation and stack buffer overflow

Can cause change of state outside of function.

5.4.7 Recurrance and the master method

5.4.8 Stack overflows

Write too many instructions to stack. can be caused by infinite recursion. eg fun x() return x()).

5.4.9 Segmentation faults and memory access restrictions

5.5 Logic

5.5.1 Propositional logic

5.5.2 Zero-order logic

Equality, inequality

5.6 Structured programming

5.6.1 Go to

5.6.2 While-do

5.6.3 If-then

5.6.4 For each loops

5.6.5 Loop x times

5.6.6 Defining and calling functions

5.7 Decompiling

5.7.1 Decompiling

5.8 Other

5.8.1 Integer caching

if we set x =2 we can either create 2 in memory, or simply point x to 2, which is already in memory

that means if we do x=2 y=2 they have the same pointer

can also cache some other common data values, eg empty lists

makes sense if pointer is smaller in memory than value

Part III

Data structures

Chapter 6

Lists

6.1 Introduction

6.1.1 Defining lists

A sequence

6.1.2 Nested lists

6.2 Read operations on lists

6.2.1 The match operation

6.2.2 The read operation

A sequence.

6.3 Write operations on lists

6.3.1 The pop operation

6.3.2 The insert operation

Chapter 7

Characters and strings

7.1 Strings

7.1.1 Encodings

Can represent same information in multiple ways?

7.1.2 American Standard Code for Information Interchange (ASCII)

7.1.3 UTF (Unicode)

7.1.4 Sort

strings as data type. similar to array? but diff functions. eg find "cat" in "concatenate"

can subinstr. upper. lower. proper. strpos

can regex. section on regex in data types?

character: number maps to character

Chapter 8

Objects

8.1 Introduction

8.1.1 Keys and values

8.1.2 Classes

8.2 Representing objects

8.2.1 Representing a single object

8.2.2 Null in objects

8.2.3 Representing a class with a multiple array (ie 2d)

8.2.4 Representing a class with a single array (ie 1d)

8.3 Functions with objects

8.3.1 Creating new objects

8.3.2 Getting values by field

8.3.3 Adding fields

8.3.4 Changing values in fields

8.4 Hierarchies of objects

8.4.1 Inheritance

8.5 Linked lists

8.5.1 Single linked lists

motivation. don't need to change who list to add things

8.5.2 Doubly linked lists

successor, predecesor and key, null if nec

8.5.3 Functions on linked lists

read, pop, add

8.6 Trees

8.6.1 Trees

8.7 Object-Oriented Programming (OOP)

8.7.1 Object-Oriented Programming (OOP)

all variable types are objects. inc integers, floats, lists etc

Chapter 9

Databases

9.1 Database operations

9.1.1 Joins

Have index common to two tables

Cross join

Keep all columns. can name [table name].[column name] to preserve any differences. including for index (could be missing for some)

Not really matched if original n length, and other m, then new is n*m length. so all combinations of matches are kept

Inner join

Drops those where no match. means data dropped in index missing

Any predicate, equi join, equality predicate

Outer join

Keep data if none matches. left outer join for one table, right outer join for other, or full outer join

9.1.2 Insert

9.1.3 Select

9.1.4 Update

9.1.5 Delete

9.1.6 Call

Chapter 10

Queues

10.1 Queues

10.1.1 Sort

queues as a type of data structure lifo. fifo. priority (ordering function)

Chapter 11

Pointers

11.1 Pointers

11.1.1 Sort

pointers? addresses? as variables? in stack?

Part IV

High-level programming

Chapter 12

Imperative and functional programming

12.1 Introduction

12.1.1 Imperative programming

In imperative programming, we say exactly what we want to happen. Each step changes the state.

12.1.2 Functions

We can call a function: $y = f(x)$

Here x are the local variables to be used.

However the global state may also affect the outcome, so we have:

$y = f(x, z)$

As a result, calling the same function twice with the same inputs can have different outputs.

Side effects. If a function modifies the state outside of its local variables it has side effects.

12.1.3 Functional programming

With functional programming, we write functions to be called. These functions should not depend on the state, outside of local variables.

12.1.4 Side effects

If we remove side effects from all functions then functional programming has no global variables which could affect the output.

Chapter 13

Object-Oriented Programming

13.1 Introduction

13.1.1 Introduction

in objects, OOP. essentially, all variable types are objects. inc integers, floats, lists etc

Chapter 14

Emulation

14.1 Low-level emulation

14.2 High-level emulation

Part V

Algorithms

Chapter 15

Algorithms

15.1 Efficiency

15.1.1 Algorithmic efficiency

An algorithm takes memory and time to run. Analysing these characteristics of algorithms can enable effective choice of algorithms.

Complexity is described using big-O notation. So an algorithm with parameters θ would have a time efficiency of $O(f(\theta))$ where $f(\theta)$ is a function of θ.

Generally we expect $f(\theta)$ to be weakly increasing for all θ. As we add additional inputs, these would not decrease the time or space requirements of the algorithm.

An algorithm which did not change complexity with inputs would have a constant as the largest term. So we would write $O(c)$.

An algorithm which increase linearly with inputs could be written $O(\theta)$.

An algorithm which increase polynomially with inputs could be written $O(\theta^k)$.

An algorithm which increased exponentially could be written $O(e^\theta)$.

Complexity can differ between worst-case scenarios, best-case scenarios and average case scenarios.

We can describe logical systems by completeness (all true statements are theorems) and soundness (all theorems are true). We have similar definitions for algorithms.

An algorithm which returns outputs for all possible inputs is complete. An algorithm which never returns an incorrect output is optimal.

15.1.2 Big-O and little-o recap

15.1.3 Time efficency

15.1.4 Space efficiency

15.1.5 Verifying answers

NP NP-hard NP-complete

15.1.6 Decision problems

Return yes or no.

15.2 Calculating the cost of an algorithm

15.2.1 Instruction costs

15.2.2 Efficiency of loops

number of times each instruction called

15.2.3 Big-O recap (take from maths)

15.2.4 Efficiency of functions with arguments

best case, worst case

15.3 Correctness

15.3.1 Correctness

An algorithm is correct if it produces the expected output for each input.

15.3.2 Partial and total correctness

An algorithm is only partially correct if may not terminate. Otherwise it is totally correct.

15.3.3 Formal verification

15.3.4 Model checking

Model checking allows the formal verification of algorithms with finite inputs. test every possible input.

15.3.5 Deductive verification

Check the parts of the algorithm using theorem provers.

Chapter 16

Algorithmic complexity

Part VI

Algorithmic paradigms

Chapter 17

Dynamic programming

17.1 Dynamic programming

17.1.1 Dynamic programming

Dynamic programming is similar to divide and conquer algorithms, in that both solve sub-problems.

However, if dynamic problems, the sub-problems overlap.

17.1.2 Hamilton-Jacobi-Bellman equation

17.1.3 Policies

A policy maps the state onto the action

$a_t = \pi(s_t)$

The policy does not need to change over time, as discounting is constant. That is, if the policy should be different in future, it should be different now.

The policy affects the transition model, and so we have P_π.

Optimal policy

There exists a policy that is better than any other policy, under any starting state.

There is no closed form solution to finding the optimal policy.

There are instead iterative methods.

17.1.4 Bellman equations

We breakdown the value function into an immediate reward, and the discounted value function of the next state.

This is because the expectation function is linear.

$v_\pi(s) = R_{s,\pi(s)} + \gamma \sum_{s'} P_{s,\pi(s)}(s') v_\pi(s')$

We can write this in matrix form.

$v_p i(s) = r_\pi + \gamma P_\pi v_\pi(s)$

We can then solve this:

$v_p i(s) = (I - \gamma P_\pi)^{-1}) r_\pi$

This depends on the starting state.

Chapter 18

Parallel processing

18.1 Parallel processing

18.1.1 MapReduce

Framework for parallising problems.

Say we have text and we want to do a word count. We can split the text into multiple subsections, do a word count on each and add these up.

MapReduce can do this.

1: Splitting

The main server splits the database into subsections.

For example, the text document could be divided into M chunks, to be split across different servers.

The master server can then pass data to be Mapped when they have capacity. Each server could run Map on a large number of documents.

Each document here is stored as a key value pair. The value here would be the text, and the key would identify the document.

$\{2 : "AppleAppleCarrot"\}$

2: Mapping

Rather than simply doing a word count on the subsections, we do a more complex, but more efficient approach.

We map the key value pair to a list of key value pairs. In our example we would map this to identifiers for each word.

The output here would be

$[\{"Apple" : 1\}, \{"Apple" : 1\}, \{"Carrot" : 1\}]$

The Map function will differ depending on the use case.

3: Shuffling

We now want to combine the key value pairs. There are R keys from the intermediate output of the Map program.

For each of the R keys, the master server assigns a server to reduce the key.

The master server then notifies the Reduce worker of the location of each relevant document output from the Map process. The Reduce server then copies the key value pair.

The Reduce worker now has an input of key value pairs. In our example one would have received:

$\{"Apple" : 1\}$

$\{"Apple" : 1\}$

It would also have received other instances from other servers.

The Reduce worker can then combine these to form the following:

$\{"Apple" : [1, 1]\}$

4: Reduce

Once all the data has been mapped, the data can be reduced.

In our example this involves creating:

$\{"Apple" : 2\}$

The Reduce function will differ depending on the use case.

5: Output

The outputs of each reduce can then be shared back to the main server.

We now have our word count.

Chapter 19

Divide and conquer algorithms

19.1 Divide and conquer

Chapter 20

Greedy algorithms

20.1 Greedy algorithms

Part VII

Elementary algorithms

Chapter 21

Sorting and searching lists

21.1 Sorting algorithms

21.1.1 Sorted lists

There can be a total ordering on elements in a list.

We want to return a list such that only the ordering is changed.

$\forall nm[list[n] > list[m] \leftrightarrow n > m]$

21.1.2 Checking a sortable list

21.1.3 Sorting algorithms

Efficient in time and memory

Popular algorithms are:

- Merge sort
- Quick sort
- Heap sort

21.1.4 Sorting linked lists

21.2 Search algorithms

21.2.1 Getting the max and min

21.2.2 Identifying the location of an element in a list

Chapter 22

Tree search

22.1 Search algorithms

22.1.1 Search algorithms

A search algorithm takes a grid and identifies a path from a start point to an end point. Each node in the grid has connections to other nodes, with costs of moving between nodes.

22.1.2 Types of nodes in a search algorithm

In each search algorithm there are three types of nodes: unexplored nodes, explored nodes and frontier nodes. At the start of the algorithm the start node is explored, each node connected to the start node is a frontier node, and all other nodes are unexplored nodes.

When an algorithm explores a frontier node, it is added to the explored nodes, and all new nodes are added to the frontier nodes.

22.1.3 Travelling salesman problem

22.1.4 Shortest path problem

22.2 Tree search algorithms

22.2.1 Breadth-first search

A breadth-first search operates First-in First-out (FiFo). That is, it selects the oldest frontier node. This results in a broad, rather than a deep search. Once all branches have been explored, the algorithm will move deeper. Path cost is not considered in this algorithm.

Informed: No

Time: $O(b^d)$

Space: $O(b^d)$

Complete: Yes

Optimal: Picks the shallowest solution. Optimal of path costs are identical.

22.2.2 Depth-first search

A depth-first search operates Last-in First-out (LiFo). That is, it selects the newest frontier node. This results in a deep, rather than a broad search. Once the maximum depth has been reached, the algorithm will move towards breadth. Path cost is not considered in this algorithm.

May not find optimal solution, but is linear in space

Informed: No

Time: $O(b^m)$

Space: $O(bm)$

Complete: Yes

Optimal: No

22.2.3 Depth-limited search

Depth-first search with a limit. This is useful if we know the solution is shallower than limit l.

Informed: No

Time:

Space:

Complete:

Optimal:

22.2.4 Iterative deepening depth-limited search

Does a depth-limited search to a layer L, increases the layer and starts again. The repeats are a waste, but earlier layers are much cheaper.

Informed: No

Time:

Space:

Complete:

Optimal:

22.3 Search algorimths with different costs

22.3.1 Uniform cost search

Modify BFS to prioritise cost not depth. expand node with lowest path cost. could be "deep".

This is the same as Dijkstras algorithm.

Can do this in algo by using heaps

Informed: No

Time: $O(b^?)$

Space: $O(b^?)$

Complete: Yes

Optimal: Yes

Chapter 23

Tree search with heuristics

23.1 Search algorithms with heuristics

23.1.1 Greedy search

Find absolute distance from goal for each node. choose node with shortest distance.

Cost of each is $f(n) = h(n)$, where $h(n)$ is the heuristic cost of node n.

23.1.2 A* search

If the heuristic is admissible, then a* is optimal. Intuitively because the the heuristic steers away from any suboptimal solutions.

Admissible? For all nodes n , h(n)¡=h*(n). where h* is true cost

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

$g(n)$ is the cost to reach n from the current position.

Informed: Yes

Time: Exponential

Space: Big, all nodes kept in memory

Complete: Yes

Optimal: Yes, if the heuristic is admissible

23.1.3 Iterative deepening A*

Chapter 24

Constraint Satisfaction Problem (CSP)

24.1 Introduction

24.1.1 Constraint Satisfaction Problem

A CSP problem is one where we don't care about the path, we just want to identify the goal state.

For example, solving a sudoku

24.1.2 Defining a CSP

A CSP has:

- Variables X_i.
- Domain for each variable D_i.
- Constraints C_j.

In a CSP there are a range of variables each with a domain. There are on top constraints on combinations of values.

A solution does not violate any constraint.

To solve, start with no allocations of variables. successor function assigns a value to an unassigned variable. goal test

Use heuristic minimum remaining value MRV: choose variable with fewest remaining legal values

Least constraining value: choose item in domain which constrains the least other moves

Forward checking. keep track of remaining legal moves for each variable. terminate if none left

After each move, update legal moves for each

Implement all this with recursive backtrack function, which returns a solution or failure. This is a depth first search

24.1.3 Arc-consistency

X-Y is arc consistent if all of domain of X is consistnet with some value of Y.

24.1.4 Node-consistency

X is node consistent if all of domain satisfies all unary constraint.

24.1.5 Path-consistency

Arc consistency for additional variables.

24.1.6 Constraint propagation

Constraint propagation can be used to prevent bad choices

We can check for:

- node-consistency
- arc-consistency
- path-consistency

24.1.7 AC-3

AC-3 algorihm makes a CSP arc-consistent

Take all arcs.

It may be possible to break the problem down into sub problems, making the problems much easier to solve.

Can do this before/after other algorithm.

24.1.8 Constraint Satisfaction Problem

Introduction

For active learning, only need to update covariance matrix? just needed to select one with highest variance

Active learning is greedy algorithm to reduce entropy

As we get more info, our posterior becomes our new prior

If we can pick observations to use to update model, can use those with biggest variance

Can be useful if getting y is expensive. requires experiement etc

4 steps:

- Form $p(y_0|x_0, y, X)$ for all unmeasured x_0.
- Choose x_0 with the largest σ_0^2 and observe y_0
- Updated the parameters with this.
- repeat

$\sigma_0^2 = \sigma^2 + x_0^T \Sigma x_0$

Updating Sigma and mu for bayesian linear:

$\Sigma = (\lambda I + \sigma^{-2}(x_0 x_o^T + \sum_{i=1}^n x_i x_i^T))^{-1}$

$\mu = (\lambda\sigma^2 I + x_0 x_0^T + \sum_{i=1}^n x_i x_i^T)^{-1}(x_0 y_0 + \sum_{i=1}^n x_i y_i)$

Once we have an x_0 we can easily get μ_0 by calculating $x_0^T \mu$. Multiplying by the mean weights.

We can also get the variance of the estimate:

$\sigma_0^2 = \sigma^2 + x_0^T \Sigma$

Part VIII

Numerical methods

Chapter 25

Floating point maths

25.1 Introduction

25.2 Square roots

25.3 Finding roots of linear equations

25.4 Finding roots of non-linear equations

25.5 Numerical integration

25.6 Trigonomic functions

25.7 Pi

25.8 e

Chapter 26

Discrete maths

26.1 Identifying primes

26.1.1 Identifying primes

different to factorising. We don't care what the actual factors are, just see if it's prime

26.1.2 Fermat's primality test

Fermat's little theorem recap

Fermat's primality test

From Fermat's little theorem we know

$a^{n-1} = 1 mod(n)$

Where a is an integer and n is prime.

26.2 Integer factorisationn

26.2.1 Trial division

We have x

Divide by numbers between 2 and x

Only need to go to sqrt x

Don't need to divide by even numbers other than 2

algorithm for checking if number is a prime

loop up dividing number from 2

if divides, add factor list and divide target number by that

stop when i reaches number

eg for 45

divide 2? no

divide 3? yes :¿ 15

divide 3? yes :¿ 5

divide 4? no

divide 5? yes :¿ 1

6¿1 so stop

number is prime if list just contains target

don't have to worry about including non primes in list, as will already have divded by that amount

26.2.2 Fermat's method

Identify the integer as the difference of two squares, and use this.

$x = a.b$

We use the midpoint of the two as $c = \frac{a+b}{2}$

This only works for odd numbers. If we have

The we have:

- $a = c + d$
- $b = c - d$
- $x = (c + d)(c - d)$
- $x = c^2 - d^2$

We can test this by trying a to get $a^2 - x$, and seeing if this is a square number.

Chapter 27

Linear algebra

27.1 Linear operations

27.1.1 Representing matrices

27.1.2 Vectors and matrices

27.1.3 Addition

27.1.4 Multiplication

27.1.5 Inverse

27.1.6 Transpose

27.1.7 Scalar multiplication

27.1.8 Matrix decomposition

27.1.9 Broadcasting

27.1.10 Broadcasting

Loosen standards, can do addition subtraction if one matrix is $1 \times n$.

27.2 Linear programming

27.3 Linear programming with integers

Chapter 28

Lossless compression

28.1 Lossless compression

28.1.1 Compression rates

28.1.2 Run-length encoding

eg 12W6RABC4D is WWWWWWWWWWWWWRRRRRRABCDDD

Part IX

Optimisation

Chapter 29

Unconstrained optimisation

29.1 Gradient descent

29.1.1 Gradient descent

29.1.2 What is gradient descent?

Rather than solve a normal equation, gradient descent takes the loss function, and takes the derivative of the loss function with respect to each parameter.

Small adjustments are then made to the parameters, in the direction of the steepest derivative, resulting in better parameters.

As derivative term gets smaller, convergance happens. The largest changes to the parametres occurs early on in the algorithm.

Can stop if not lowering by much

29.1.3 Local minima

Gradient descent is not guaratneed to arrive at a global minimum. For some loss functions, there will be multiple local minima, and gradient descent can end up in the wrong one.

Linear regression does not have this issue.

As a result, when we create functions with loss functions, convextity is very important. If the loss space is convex, then we will not get stuck in a local minima.

29.1.4 Momentum gradient descent

Batch gradient descent

:= used to denote an update of variable. Used in programming, eg x=x+1.

$\theta_j := \alpha \frac{\delta}{\delta \theta_j} J(\theta_0, \theta_1)$

α sets rate of descent.

$\theta 0 := \theta 0 - \alpha / m \sum (h0(x) - y)$

$\theta j := \theta j - \alpha / m \sum (h0(x) - y) xj$

Can check if j theta increasing, means bad methodology, lower alpha

Get run for x iterations,evaluate j(theta)

Can use matrices to do each step

Can check convergence by checking cost over last 1000 or so, rather than all

Smaller learning rate can get to better solution, as can circle drain for small samples

Slowly decreasing learning rate can get better solutions

$\alpha = const1/(i + cost2)$

Do gradient descent on all samples

The standard gradient descent algorithm above is also known as batch gradient descent. There are other implementations.

Mini-batch gradient descent

Use b samples on each iteration, b is parameter, between stochastic and batch

$b = 2 - 100$ for example

Stochastic gradient descent

Do gradient descent on one (?!) sample only

Not guaranteed for each step to go towards minimum, but each step much faster

Stochastic gradient descent with momentum

The gradient we use is not just determined by the single sample, it is a moving average of past samples.

Epochs

This refers to the number of times the whole dataset has been run.

29.1.5 Adaptive learning rates (Adagrad, Adadelta, RMSProp, ADAptive Momentum (ADAM))

29.1.6 Adagrad

29.1.7 Adadelta

29.1.8 RMSProp

29.1.9 ADAptive Momentum (ADAM)

29.2 Differentiable on a single axis

29.2.1 Coordinate descent

29.3 Twice-differentiable functions

29.3.1 Algorithmic efficiency

An algorithm takes memory and time to run. Analysing these characteristics of algorithms can enable effective choice of algorithms.

Complexity is described using big-O notation. So an algorithm with parameters θ would have a time efficiency of $O(f(\theta)$ where $f(\theta)$ is a function of θ.

Generally we expect $f(\theta)$ to be weakly increasing for all θ. As we add additional inputs, these would not decrease the time or space requirements of the algorithm.

An algorithm which did not change complexity with inputs would have a constant as the largest term. So we would write $O(c)$.

An algorithm which increase linearly with inputs could be written $O(\theta)$.

An algorithm which increased exponentially could be written $O(e^{\theta})$.

Complexity can differ between worst-case scenarios, best-case scenarios and average case scenarios.

We can describe logical systems by completeness (all true statements are theorems) and soundness (all theorems are true). We have similar definitions for algorithms.

An algorithm which returns outputs for all possible inputs is complete. An algorithm which never returns an incorrect output is optimal.

29.3.2 BroydenFletcherGoldfarbShanno (BFGS) algorithm

29.4 Non-differentiable functions

29.4.1 Subgradient descent

29.4.2 Grid search

29.4.3 Hill climbing

We initialise at some point in the parameter space.

We identifify nearby alternative points in parameter space, and move to the one with the most improvement.

Movement only occurs in one parameter at a time.

29.5 Sort

29.5.1 Floating point

29.5.2 Integer

29.5.3 Tree path

29.5.4 Array

Eg 8 queens

29.5.5 Search tree

in search tree, each node has state which is used for test. could be ID of node (for path finding), path history and cost (for trav salesman)

frontier (not open list)

backward search. only possible if end state is clearly defined. eg maze. not clear if don't know eg 8 queens.

can do breadth first on them simultaneously?

problem has: initial state. actions, transition model

model $T(s,a) -> s_n + 1$. as in , given state and action, we have new state

goal test on each state

path cost for each sucessor

search tree. we expand when testing action.

open lists in unexplored notes.

loopy paths. if we go $a -> b$ don't need to go $b -> a$ because if goal, not any closer, if util, higher cost.

redunant paths. if we've already been to c, no need to explore going there from somewhere else in goal

if already been to c at lower cost, no point for util

actions is function on state.

keep explored states in open list

Chapter 30

Constrained optimisation

30.1 Linear programming

30.2 Quadratic programming

30.3 Integer programming

30.4 Convex programming

Part X

Improving knowledge

Chapter 31

Knowledge bases

31.1 Storing knowledge

31.1.1 Resource Description Framework (RDF)

Introduction

How can we store information like "Joe Blogs was born in the city London"?
Information is described as an RDF triple:

- Subject
- Predicate
- Object

Examples

"Joe Blogs was born in the city London" can be written as:
(Joe Blogs, BornCity, London)

Confidence

We can associate a confidence with each triple.

31.1.2 Knowledge bases as graphs

Introduction

We can consider each fact to be a mini graph.

For "Joe Blogs was born in the city London" we have:

Joe Blogs $\rightarrow^{wasborninthecity}$ London

31.2 Using knowledge

31.2.1 Inferring facts

Introduction

Now we add another fact:

London $\rightarrow^{isacityinthecountry}$ UK

We can use these to define a new predicate: "was born in the country" and generate the fact:

Joe Blogs $\rightarrow^{wasborninthecountry}$ UK

31.2.2 Relational learning

Introduction

Consider two facts:

Alice $\rightarrow^{IsA}$ Doctor

Bob $\rightarrow^{HasMother}$ Alice

We can consider another fact:

Bob $\rightarrow^{IsA}$ Doctor

Confidence of inferred facts

How confident should we be of this?

In practice the graph between Bob and Doctor will have many paths (qualifications)

31.2.3 Prior and posterior confidence

Introduction

If we have a new fact, and a prior, we can create a posterior condfidence on the fact.

31.3 Collecting knowledge

Chapter 32

Expert systems

32.1 Logic-based agents

32.1.1 Forward and backward chaining

Introduction

Forward and backward chaining

Has a knowledge base (domain specific content)

Has an inference mechanism (domain independent algorithms)

If we want to find out whether the knowledge base implies a statement, we can check using semantics or sytax.

Inference is similar to search. We start with the knowledge base. We can apply the inference rule to all statements which match the top line of modus ponens (or another rule).

The actions are inference rules. We add new sentences to the b

Agents can also interact with the world in order to add to the knowledge base

Convert each item in knowledge base to CNF for resolution rule.

Part XI

Applications

Chapter 33

Image manipulation

33.1 Introduction

33.1.1 Anti-aliasing

33.1.2 Rotating images

33.1.3 Layers

33.1.4 Scaling images

33.1.5 Mode 7

Chapter 34

3D modelling

34.1 Introduction

34.1.1 Rasterisation

34.1.2 Shading