Optimisation and Operations Research

Lecture 12: Algorithm Analysis and Complexity

Matthew Roughan
<matthew.roughan@adelaide.edu.au>
http://www.maths.adelaide.edu.au/matthew.roughan/notes/OORII/

School of Mathematical Sciences,
University of Adelaide

September 12, 2019

Section 1

Algorithm Analysis and Indeterminacy

Strategy for counting operations

- Break the program into blocks
  - we can just add up the cost of each block
- Look for loops
  - the cost of the "stuff" inside the block is *multiplied* by the number of times the loop runs
- Using Big-O notation simplifies our work
  - Big-O with respect to asymptotic problem *size*
  - We only have to keep track of "biggest" parts of an algorithm
  - Uniform-cost model works
    - ★ constant factors drop out, so
    - ★ we can count all operation types as $O(1)$
  - Elementary functions are also $O(1)$

Time Analysis: indeterminacy

The big problem for analysing cost is indeterminacy, *i.e.*, sometimes the code's behaviour changes depending on the values

```
if (x > 0)
  y = x + 1
else
  y = very_complicated_function(x)
end
```

Often we don't know what value of x to expect (that's might be the whole point of the program) so how should we analyse this?

Worst case analysis

- Usually look at worst case performance
  - because we are conservative
  - because this is often easier to calculate
- In the example above, the worst case is the complicated function, so we use its complexity
- The hard cases involve things like
  - a loop where the number of iterations is variable
  - complicated logic and calculations that depend on inputs

  e.g., Simplex
- Let's have a bit more of a look
- Remember we will use Big-O notation, which makes our lives easier

Section 2

Worst Case Analysis

Instances

Any (interesting) algorithm can solve many *instance* of a given problem

- *e.g.*, Simplex can solve lots of different LPs
  - any given set of objective and constraints defines an instance
- Computation time depends on the particular instance
- We usually parameterise instances by *size*
  - the size of a LP is the number of variables and constraints
- The *worst case* is the instance of a given size that takes the longest to solve
  - note this is a very specific sense of "worst"

Example 1

Evaluating x^n

- Naïvely we do this with $n-1$ multiplications, but that is slow
- Actually we use tricks like $x^n = \exp\left(n \log x\right)$
- Simple functions like $\exp(\cdot)$ and $\log(\cdot)$ are $O(1)$ (for fixed precision)
- The instance *size* is n
- The "worst case" here doesn't matter, because all calculations take the same time (with respect to x and n)

So $x^n = O(1)$

Example 2

Evaluating a degree n polynomial

$$a_n x^n + \cdots + a_1 x + a_0$$

- Assume we calculate it as written
- The instance *size* is the degree n
- The worst case instance: all of the coefficients a_i are non-zero
- In the worst case, we have to do at least n additions
- As noted above $x^n = O(1)$

So evaluating a degree n polynomial directly is $O(n)$

Example 2b

Evaluating a degree n polynomial

$$a_n x^n + \cdots + a_1 x + a_0$$

- Naïve way to evaluate is to calculate the sum as it is written
- There is a better way – Horner's algorithm

$$\Big(\big((a_n x + a_{n-1})x + a_{n-2}\big)x \cdots + a_1\Big)x + a_0$$

- Note that this is still $O(n)$, but it will typically be faster
  - Big-O notation hides constant factors

Example 3

```
if (condition)
  % sequence of statements 1 which is O(n)
else
  % sequence of statements 2 which is O(n^2)
end
```

- Assume worst case, so take the worst branch
- Algorithm is $O(n^2)$

Example 4

```
while (n > 0)
  % make n smaller by some amount x >= 1
  % steps take O(n^2) to calculate
end
```

- "Size" is n
- Assume work inside the loop takes $O(n^2)$
- The worst case is that $x = 1$ for all loops, so we have to go through the loop n times, and hence the algorithm is $O(n \times n^2) = O(n^3)$

We often look for "bounding" cases, *i.e.*, worst cases where a loop is actuated the maximum possible number of times

Example 5

Euclid's algorithm for Greatest Common Divisor (GCD)

```
function gcd(a, b) // for positive integers a,b
    while a not= b
        if a > b
            a := a - b;
        else
            b := b - a;
        endif
    endwhile
    return a;
```

Euclid's algorithm examples

step	a	b
start	35	21
1	14	21
2	14	7
3	7	7

step	a	b
start	34	21
1	13	21
2	13	8
3	5	8
4	5	3
5	2	3
6	2	1
7	1	1

Euclids algorithm examples

- Euclid's algorithm is very old (at least 300 BC)
- In the uniform-cost model each step takes constant time, so we just have to count the number of steps
  - presumes integers of fixed size
- The number of steps is bounded by $n = \max(a, b)$, so algorithm looks like $O(n)$
- However, the worst case was discovered Émile Léger, in 1837
  - Fibonacci numbers F_N
  - $F_N \geq \varphi^{N-1}$, (Golden Ration Phi)
  - So $N - 1 \leq \log_\varphi n$

  So algorithm is $O(\log n)$
- Analysis of this by Lamé in 1844 was first case of complexity analysis

Example: Simplex

What is the computational complexity of Simplex?
Let's just look at Phase II, with n variables and n constraints

- Each pivot requires $O(n^2)$ operations (in the worst case)
  - each element of the matrix might be multiplied and/or added
- If we do k pivots, then the algorithm would be $O(kn^2)$

But what is k? It probably depends on n and m, but how?

- ? $k = O(nm)$
- ? $k = O(exp(nm))$

Example: Simplex

- we will see an example (Klee-Minty) of a Simplex problem where k is exponential, so the worst case behaviour of Simplex is exponential
- but common cases typically have iterations $k \leq 3m$
- there are (guaranteed) polynomial time algorithms for solving LP (see interior point algorithms), though these aren't necessarily much faster on many problem, its just their worst case is guaranteed to be better.

Strategy

- Break the program into blocks
  - we can just add up the cost of each block
- Look for the "biggest bits"
- Look for loops
  - the cost of the "stuff" inside the block is *multiplied* by the number of times the loop runs
- Where there are indeterminate parts, look for the worst case
  - branches (conditionals) – choose worst branch
  - loops – look for bounds
- We can often cheat
  - skip calculations for "small" bits
  - there are known complexities for many common algorithms – you can use these as blocks

Further complexity

Other measures of complexity

- Communications complexity
  - *e.g.*, how much network traffic on a parallel cluster
- Memory complexity
  - how much memory is needed for the algorithm

We still use techniques like Big-0 and worst case analysis

Others ways to assess complexity

- Average case
  - Simplex typical case is polynomial, instead of exp
- Best case

Takeaways

- Strategies and examples for calculating computational complexity of more complicated algorithms

Further reading I