Algorithmic problem-solving: Tractable vs Intractable problems

Based on Part V of the course textbook.

Algorithmic techniques

Question: Given a computational task, what sort of algorithms are available for it? What time complexities can we expect?

We have already looked at a few problem solving techniques:

1. Divide-and-Conquer
2. Greedy Strategies
3. Dynamic Programming

These are some of the important techniques, but there are others.

Techniques (1) and (2) tend to give fast (polynomial-time) algorithms. (3) can give fast algorithms, but can also be used when no fast algorithms are available.

Polynomial-time vs exponential algorithms

An algorithm is said to be polynomial-time if its worst-case time complexity is amongst:

$$O(1), O(N), O(N^2), O(N^3), \ldots$$

Whilst $O(1)$ is (very!) fast, $O(N)$ is fast, $O(N^{100})$ is slow, but we still consider these as fast algorithms and as tractable solutions to a problem.

Exponential algorithms: Algorithms that behave in the worst-case as 2^N or N^N or $N!$ (N factorial) are very slow and are considered as intractable algorithms in general.

Note: Stirling's formula (with $e = 2.71828\ldots$ the base of natural logarithms):

$$N! \approx \sqrt{2\pi N}(N/e)^N$$

Thus factorial is not only exponential, but badly so!

Why are exponential algorithms intractable?

Exponential algorithms run very slowly as the size of the input N increases and on fairly small inputs require impossible computational resources:

N	2^N	$N!$	N^N
10	10^3	4×10^6	10^{10}
20	10^6	2×10^{18}	10^{26}
100	10^{30}	9×10^{157}	10^{200}
$1,000$	10^{301}	10^{2672}	10^{3000}
$10,000$	10^{3000}	10^{36701}	10^{40000}

The age of the universe is approximately 4×10^{26} nanoseconds.

It is thought that there are approximately 10^{80} protons in the universe (the Eddington number).

What algorithms are available?

Finding how difficult a computational task is, is difficult!

Problems which appear very similar, or even variants of each other, can have very widely different time complexities.

Remember the coins problem? Some collections of coin values have fast greedy algorithms, other collections of values don't have any fast algorithms.

Where do exponential algorithms come from?

Exponential algorithms arise from various algorithmic techniques:

- Exhaustive enumeration - listing all possibilities and checking each for the solution requirement, or
- Systematic search of the space of (partial) solutions using unbounded backtracking.

An algorithmic task: Graph colouring

A graph consists of a set of nodes linked by (undirected) edges.

A colouring of a graph with k colours is an allocation of the colours to the nodes of the graph, such that each node has just one colour and nodes linked by an edge have different colours.

The minimum number of colours required to colour a graph is its chromatic number.

Applications: Graphs are usually used where ‘connectedness’ is being recorded. Colouring is about the opposite: edges are present when nodes need to be separated.

An exponential-time colouring algorithm - by enumeration

Exhaustive enumeration is the simplest approach to colouring a graph with k colours:

1. Enumerate all allocations of k colours to the nodes of the graph,
2. Check each allocation to see if it is a valid colouring.

For a graph with N nodes and E edges, there are k^N allocations of k colours to the nodes.

Checking each allocation to see if it is a valid colouring, is $O(E)$.

So this algorithm has worst-case time complexity $O(E \times k^N)$.

A graph colouring algorithm by systematic search

Here is an algorithm for k-colourability using exhaustive unbounded back-tracking:

Number the nodes $1, \ldots, N$. For each node encountered, it successively tries the colours. If all fail, it undoes the colour of the previous node and tries new colours for it. If it succeeds, it then tries to colour the next node.

```
n := 1;
while n =< N do
  { attempt to colour node n with
    next colour not tried for n }
  if there is no such colour
    then
       if n>1 then n := n-1 else fail
    else if n=N then print colouring else n := n+1
```

Time complexity: This is exponential in worst-case. Why?

Exponential algorithms: continued

In fact, all known algorithms for graph colouring (for $k \geq 3$) are exponential!

Graph colouring is an example of what is known as an NP-complete problem. The only algorithms known for NP-complete problems are exponential.

Whether NP-complete problems admit polynomial-time algorithms is one of the most important open problems in Computer Science.

There is one million US dollars available to anyone who can come up with a polynomial algorithm for graph colouring!

More on this topic (and NP-completeness) will be found in the Advanced Algorithms course in the third year.

Intractable problems: Examples

Many common computational problems are NP-complete and therefore all known algorithms are exponential:

For example, the celebrated...

Travelling salesperson problem: Find a tour of N cities in a country (assuming all cities are reachable). The tour should (a) visit every city just once, (b) return to the starting point and (c) be of minimum distance.

This is an optimisation problem - not just find a solution but find ‘the best’ solution.

Travelling salesperson: Solutions

Solution by exhaustive enumeration:

1. Starting at any city, enumerate all possible permutations of cities to visit,
2. Find the total distance of each permutation and choose one of minimum distance.

There are $(N-1)!$ permutations for N cities (the starting city is arbitrary), so this is an exponential solution.

There are many other approaches to the travelling salesperson problem, for example using dynamic programming. All are exponential in worst-case (but there are faster approximate algorithms).

Knapsack problems

A Knapsack problem requires us to fit a number of items of different sizes into a container ('knapsack'). We are asked to do this optimally: i.e. fill the container as much as possible (or, equivalently, minimise the unused space).

More generally, the items have values as well as size/weight and we are asked to maximise the total value in the knapsack.

This is the next laboratory exercise.

Solutions: Some Knapsack problems have polynomial (indeed linear) solutions, others have only exponential solutions.

Job scheduling, rostering and timetabling

General problem: A collection of tasks are to be undertaken by allocating them to slots. Slots may be time periods, machines, rooms, people's availability etc. or combinations of these.

Usually, the allocation is subject to constraints:

1. Compatibility constraints: which tasks fit which slots.
2. Scheduling constraints e.g. temporal orders: some task may have to be performed before others, some may or may not be allowed to be performed simultaneously, etc.
3. Resource constraints: are resources unlimited or limited?
4. Optimisation requirements: We may require solutions that maximise or minimise given measures. For example, we may require that all tasks are completed in a minimum time, or that the allocation distributes tasks as evenly as possible.

The problem is to allocate tasks to slots satisfying the constraints.

Scheduling: Examples

(1) Scheduling algorithms in multi-tasking operating systems by which threads, processes or data flows are given access to system resources. Constraints include effective load balancing and achieving a required quality of service.

(2) Industrial processing: Combining jobs and their scheduling to achieve required outputs.

(3) Rostering: For example, constructing a roster of buses and drivers to provide a bus service: Your Software Engineering laboratory exercise.

(4) Timetabling: Allocating students, staff, classes, rooms and times to provide a timetable.

Algorithms for scheduling

The wide variety of such tasks is matched by the variety of algorithmic solutions.

- Some have linear or polynomial solutions, often using simple allocation methods: For example, first-come-first-served allocations, or greedy methods.
  The simple form of your SE laboratory exercise has such a solution.
- However, most scheduling problems have only exponential solutions in general.
  The general case of timetabling is such a problem: exhaustive enumeration is one approach.
  Graph colouring is a method of solving some job scheduling problems: Suppose the nodes are tasks. Two jobs are linked by an edge if they conflict i.e. cannot occupy the same slot. A colouring with k colours schedules the jobs into k slots without conflict... and this problem has only exponential algorithms.

Heuristics

Heuristics: Heuristics are rules that can be used in decision making (for example, in exploring a space for solutions). They are intended to reach optimal solutions, but are not guaranteed to do so.

Heuristics are useful when exhaustive searches are exponential-time, and may reduce the search to polynomial-time, but achieve only approximate solutions in general.

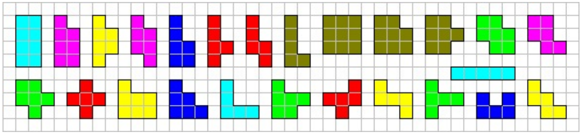
Puzzles

Most puzzles have only exponential algorithms to solve them.

That is what makes puzzles interesting!

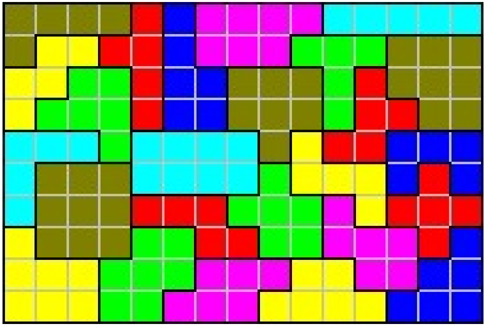
In general, the only solutions are by exhaustive search based on unbounded backtracking.

Example: Polyominoes



Problem: to fit these into a 10×15 rectangle.

One solution



[Website: http://www.iread.it/lz/polyedges.html]