Advanced Algorithms I

Todd Davies

December 22, 2015

Introduction

This course unit has two objectives. The first is to introduce the student to a range of advanced algorithms for difficult computational problems, including matching, flow networks and linear programming. The second objective is to outline the mathematical techniques required to analyse the complexity of computational tasks in general. There are two pieces of assessed coursework, and an exam at the end.

Aims

This unit provides an advanced course in algorithms, assuming the student already knows algorithms for common computational tasks, and can reason about the correctness of algorithms and understand the basics of computing the complexity of algorithms and comparing algorithmic performance.

The course focuses on the range of algorithms available for computational tasks, considering the fundamental division of tractable tasks, with linear or polynomial-time algorithms, and tasks that appear to be intractable, in that the only algorithms available are exponential-time in the worst case.

To examine the range of algorithmic behaviour and this fundamental divide, two topics are covered:

- Examining a range of common computational tasks and algorithms available; We shall consider linear and polynomial-time algorithms for string matching tasks and problems that may be interpreted in terms of graphs. For the latter we shall consider the divide between tractable and intractable tasks, showing that it is difficult to determine what range of algorithms is available for any given task.
- Complexity measures and complexity classes: How to compute complexity measures of algorithms, and comparing tasks according to their complexity. Complexity classes of computational tasks, reduction techniques. Deterministic and non-deterministic computation. Polynomial-time classes and non- deterministic polynomial-time classes. Completeness and hardness. The fundamental classes P and NP-complete. NP-complete tasks.

Additional reading

Title	Author	Year
Core Algorithm design: foundations, analysis and internet examples	Goodrich, Michael T. and Roberto Tamassia	2002
Introduction to the theory of computation (3rd edition)	Sipser, Michael	2013

Contents

1 Algorithmic Wisdom

Lectures 2 & 3

The first two lectures of the course describe different types of algorithms, what is computable, where to optimise algorithms, asymptotics and heuristics.

1.1 Different types of algorithms

There are three types of algorithms that are mentioned:

> Note:
>
> You should know all of this from `COMP26120`.

Divide and conquer

These algorithms continually break a problem down into smaller parts, which are easier to solve, until eventually, the problems are trivial and easily solved. This is often used when the data you're operating on is in a recursive datastructure such as a tree. If you're writing an algorithm to find how many nodes there are in a tree, then you could use divide and conquer:

```
int countTreeSize(tree) {
int size = 1;
if (tree.left) size += countTreeSize(tre
if (tree.right) size += countTreeSize(t
return size;
}
```

As you can see from the example, divide and conquer algorithms are usually recursive.

The divide and conquer technique can be applied to graphs, but in order to do this, you must keep track of which nodes you've visited with a flag on each node. If we wanted to count the nodes in a graph, we could do:

```
int countGraphSize(graph) {
if (graph.visited) return 0;
graph.visited = true;
int size = 1;
for (child in graph) {
  size += countGraphSize(child);
}
return size;
}
```

Mutual Recursion

Mutual recursion describes an algorithm that operates on data where one type of data can reference another, and the other can reference it. The example given is that of statements and expressions in programming languages; statements contain expressions, and expressions can also contain statements. Parsing such a structure might involve two algorithms that recurse on each other!

Dynamic Programming

Dynamic programming exploits the fact that when some problems are broken down into smaller sub-problems, some of the sub-problems are identical. Dynamic programming algorithms start from the very smallest sub-problems and build up to the final solution, and usually

cache results to sub-problems in a table so that work is not done twice.

1.2 Computability

There are many different definitions of computability, including lambda calculus, Turing machines, rewriting rules, random access machines and (many) more. The idea that relates all of these things, is that they all have the same capabilities. That is to say that if you can compute something using one of these ideas, then you can also compute it on all the others too.

There are also a class of 'alternate' computing mechanisms, such as quantum computers and neural computers. These ideas have the potential to compute things that a Turing machine (or its equivalents cannot), but they are significantly harder to build, and functional implementations do not exist yet.

1.3 Asymptotics and optimisation

When you have to get a computer to perform a task, implementing a simple algorithm first is a good idea, since you will at least have something to demonstrate to people, and you will gain a good understanding of the problem at hand. However, simple algorithms are often slow; how should we evolve our implementation to be as fast as we need it to be?

Profiling can tell you where your code is spending most of its time. Sometimes your algorithm will be really fast, and the processor will spend most of its time waiting for IO to give it more data; this is often the case with GPU computation.

Assuming you find some CPU bottleneck in your code, before you spend hours making it faster, consider whether it is worth the effort. If this part takes up 10% of your runtime, and you make it twice as fast, your program will only run 5% faster. This is an example of the Law of Diminishing Returns.

As well as optimising specific parts of an algorithm, you also should consider its asymptotic run time. An algorithm that runs in $O(n^2)$ time is probably going to be better than one that runs in $O(nlog(n))$ time. However, this isn't always the case; some algorithms (often ones with good asymptotic run times) take a long time to set up, usually when you have to transform the data into some different datastructure. If you are running your algorithm on a small amount of data, then an algorithm that you can run on your data *as is* might outperform a fancier algorithm that you have to invest more overhead in.

Note:

Sometimes a good solution is to use different algorithms depending on the input. If there are only a few cases that produce worst-case performance, you could even hard-code solutions to those!

The average case runtime of a algorithm is also important. Haskell uses a type checker that runs in $O(2^{n^n})$ time in the worst case, but for every program that isn't made specifically

to mess with the compiler, runs in linear time.

2 Graphs

A graph is a pair $G = (V, E)$ where V is a finite set, and E is a set of pairs between items in V. Elements in V are *vertices* or *nodes*, and elements in E are *edges*.

Different mathematicians have different rules about what exactly can go in a graph. For the purposes of this course, the graphs shown in Figure 1 aren't allowed.

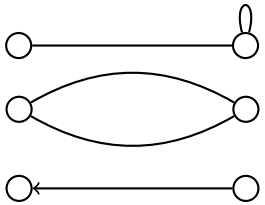

Figure 1: Self loops, duplicate edges and arrows are not allowed.

A **directed graph** is just like a normal graph, except the edges do have arrows. The only mathematical difference is that the set E is a set of ordered pairs.

The *degree* of a node in a graph, is the number of edges that are adjacent to (touching) it. If the graph is directed, then it is the number of edges originating from the node.

A weighted graph is one where each edge is associated with a value representing its weight. The length of a path between nodes is simply the sum of the edge weights connecting the nodes.

2.1 Connectivity

A node a is **reachable** from another node b if there is some sequence of nodes connected by edges that go from the a to b.

A graph where every node is reachable from every other node is **connected**. A strongly connected graph is a **directed graph** where each node is reachable from each other node.

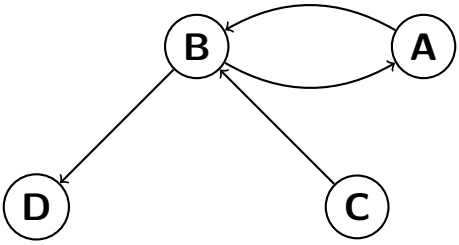

Figure 2: This graph is not strongly connected, but the subgraph with only nodes A and B is. If we were to remove the arrows, then the graph would be connected.

2.2 Representing graphs

We can store graphs in computer memory in two ways:

Adjacency List
There are two different kinds of adjacency list, the first is like so:

```
public class Graph {
  List<Vertex> nodes;
  List<Edge> edges;
```

```
}
public class Edge { Vertex a, b; }
public class Vertex { List<Edge> outl
```

Here, we keep a list of all the nodes, and a list of all the edges. From any edge, we can see what nodes it connects, and from any node, we can see what edges it connects.

The other type of adjacency list is a bit simpler, but less efficient in some cases:

```
public class Graph<T> {
  Set<T, List<T>> adjList;
}
```

Adjacency Matrix

Here, a matrix indicates whether there is an edge between two nodes:

	A	B	C	D
A	0	1	0	0
B	1	0	0	1
C	0	1	0	0
D	0	0	0	0

This adjacency matrix represents the graph in Figure 2. You can see that it is fairly wasteful in terms of memory $O(|E| * |V|)$, though with bit arrays, it has a very low constant overhead.

2.3 Graph algorithms

You will have covered some of the algorithms featuring here in previous courses, or perhaps seen them in the wild. They are however interesting and useful, so it's worth a recap even if they're not new! I've ordered these in roughly increasing order of mental strain.

2.3.1 Depth first search

Depth first search (DFS) is an algorithm to find a node in a graph starting from another node. It works on both directed and undirected graphs, and runs in $O(|V| + |E|)$ (linear) time. The psudo code looks like this:

```
Node dfs(Node haystack, Node needle) {
  Stack<Node> toVisit = new Stack<>();
  Set<Node> visitedNodes = new Set<>();
  toVisit.push(haystack);
  while(!toVisit.isEmpty()) {
    Node node = toVisit.pop();
    if (visitedNodes.contains(node)) continue
    visitedNodes.add(node);
    if(node.equals(needle)) {
      return needle;
    } else {
      for(Node child : node.children) {
        toVisit.push(child);
      }
    }
  }
  return null;
```

```
}
```

A Breadth First Search is the same, except you use a `Queue` instead of a `Stack`.

> Note:
>
> To see if a graph is connected, do a depth first search as in the example code, but don't stop when you find a needle, only stop when the `toVisit` stack is empty. If `visitedNodes` contains all of the nodes in the graph, then the graph is connected.

Depth first search also lets you find if one node is reachable from another in linear time, and also if a graph is connected in linear time too, with a few modifications.

We can also find out if a directed graph is strongly connected in $O(|V| + |E|)$ time using Tarjan's algorithm, which we'll see later.

2.3.2 Dijkstra's algorithm

Dijkstra's algorithm finds the undirected shortest path between two nodes in a graph. Here is the psudo-code:

```
function Dijkstra(Graph, source):
  create vertex set Q

  for each vertex v in Graph:           // Initia
    dist[v] = INFINITY                  // Unknow
    prev[v] = UNDEFINED                 // Previo
    add v to Q                          // All no
```

```
dist[source] = 0                                  // Distan

while Q is not empty:
  u = vertex in Q with min dist[u]                // Source
  remove u from Q

for each neighbour v of u:                         // where
  alt = dist[u] + length(u, v)
  if alt < dist[v]:                               // A shor
    dist[v] = alt
    prev[v] = u

return dist[], prev[]
```

Listing 1: Dijkstra's algorithm (from Wikipedia)

If we use a Fibonacci heap, for the priority queue, then the runtime of Dijkstra's algorithm is $O(|E| + |V|log(|V|)$. If we use a normal heap, then the runtime is $O(|E|log(|V|))$.