10 A: How Difficult Can Problems Be?

CS1102S: Data Structures and Algorithms

Martin Henz

March 27, 2009

Generated on Wednesday 1st April, 2009, 17:07

1. Problem Difficulty
2. Greedy Algorithms

1. Problem Difficulty
   - Algorithmic Problems
   - Undecidability of the Halting Problem
   - NP-Complete Problems

2. Greedy Algorithms

How Difficult Can Problems Be?

Examples

- Finding the smallest element in a linked list: $O(N)$
- Inserting an element into a heap: $O(\log N)$
- Sorting an array of items using comparisons: $O(N \log N)$
- Printing all sequences of N binary digits: $O(2^N)$

Algorithmic Problems

Given Input

Clear description of input data

Required Output

Description of what a solution constitutes

Algorithmic Solution

Description of a process how to arrive at the required output, given any legal input

Questions

- Are all algorithmic problems solvable?
- Are all algorithmic problems solvable in polynomial time? Is there a k such that there is an $O(N^k)$ algorithm for the problem?
  Clearly no! The output can have exponential size!
- If we do not have a polynomial algorithm, can we always prove that there is none?

Halting Problem

Definition (Halting Problem)

Given a description of a program and a finite input, decide whether the program finishes running or will run forever, given that input.

Undecidability of the Halting Problem

Theorem (Undecidability of the Halting Problem)

The Halting Problem is undecidable; there cannot exist a program that returns `true`*, if and only if a given function* `f` *terminates when applied to a given value* `x`*.*

Proof of the Undecidability of the Halting Problem

Assume that there is a Java function `halts` that when applied to a representation `f'` of a Java function `f`, and a Java object `x` returns `true` if `f(x)` terminates, and `false` if `f(x)` does not terminate. Such a function `halts` exists if and only if the Halting Problem is decidable.

Proof of the Undecidability of the Halting Problem

With the assumption of the existence of `halts`, let us construct a Java function `strange` as follows:

```
boolean strange(w’) {
  if (halts(w’,w’)
    while (true);
  return true;
}
```

Such a function `strange` can surely be written, if `halts` exists.

Proof of the Undecidability of the Halting Problem

```
boolean strange(w’) {
  if (halts(w’,w’)
    while (true);
  return 0;
}
```

What will `strange(strange’)` return?

Proof of the Undecidability of the Halting Problem

Conclusion: `strange` cannot exist, and therefore `halt` cannot exist.

The Halting Problem is undecidable.

Example: Hamiltonian Cycles

Given Input

An undirected connected graph

Desired Output

A path that starts and ends in the same vertex and contains all other vertices exactly once.

No efficient algorithm known

We do not know if there is a k such that the problem can be solved in $O(N^k)$.

Our Last Question

Question

If we do not have a polynomial algorithm, can we always prove that there is none?

Answer

No: we cannot (at this moment) prove that there is no polynomial algorithm for the Hamiltonian cycle problem

Verifying Solutions

Example: Hamiltonian cycle problem

If we have a candidate of a solution to the problem, we can easily check that it is correct.
Simply check that the last vertex in the cycle is that same as the first, and that every vertex of the graph is contained in the cycle.

Definition

We call the class of problems for which solution candidates can be checked in polynomial time *NP*.

NP-Complete Problems

Other problems in NP

- Boolean satisfiability problem (SAT)
- Graph coloring problem
- Clique problem

Reducibility

It turns out that all the mentioned problems can be transformed into each other in polynomial time! Thus if we can solve one, we can solve all.

NP-Complete Problems

The class of problems that can be transformed into the Hamiltonian path problem in polynomial time is called the class of *NP-complete problems*.

Nonpreemptive Scheduling

Input

A set of jobs with a running time for each

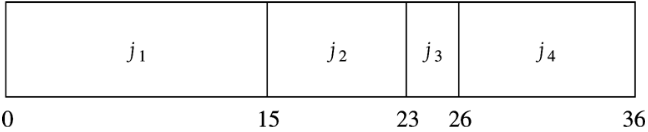
Desired output

A sequence for the jobs to execute on on single machine, minimizing the average completion time

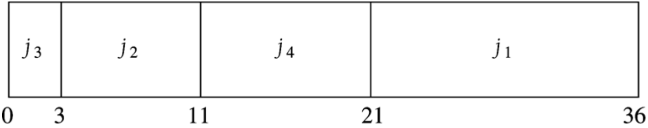
Example

Job	Time
j_1	15
j_2	8
j_3	3
j_4	10

Some schedule:



The optimal schedule:



The Optimal Solution

Theorem

Ordering the jobs in increasing length leads to a schedule with minimal average completion time.

Proof

$$\begin{aligned} C &= \sum_{k=1}^{N} (N - k + 1) t_{i_k} \\ &= (N+1) \sum_{k=1}^{N} t_{i_k} - \sum_{k=1}^{N} k \cdot t_{i_k} \end{aligned}$$

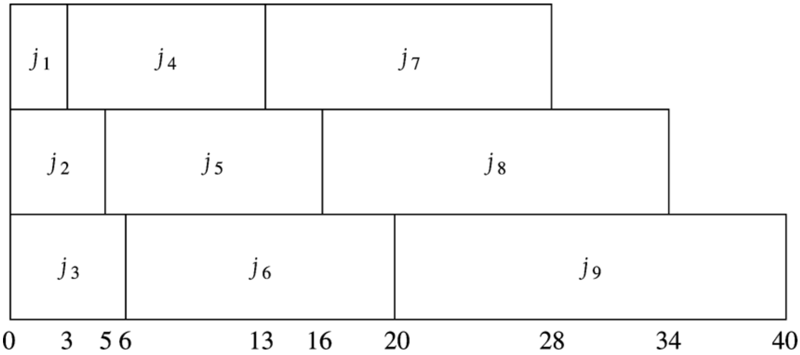
If $x > y$ such that $t_{i_x} < t_{i_y}$, then by swapping x and y the second sum increases, thus the overall cost decreases.

The Multiprocessor Case

N processors

Now we can run the jobs on N identical machines. What is a schedule that minimizes the average completion time?

Example



A "Slight" Variant

Miniming *final* completion time

If we want to minimize the *final* completion time (completion time of the last task), the problem becomes NP-complete!