1. Computational Complexity

Computational complexity theory provides a mathematical framework that is able to explain why some problems are easier to solve than the others. We briefly sketch some of the main points of this theory. Detailed exposition of this topic can be found in the books by Garey & Johnson (1979) and Papadimitriou (1994).

Most of scheduling problems are optimisation problems, i.e., we look for a schedule that minimises a certain objective function. An algorithm is a step-by-step procedure for solving a computational problem. For a given input, it generates the correct output after a finite number of steps. The time complexity or the running time of an algorithm expresses the total number of elementary operations, such as additions, multiplications and comparisons, for each possible problem instance as a function of the size of the instance.

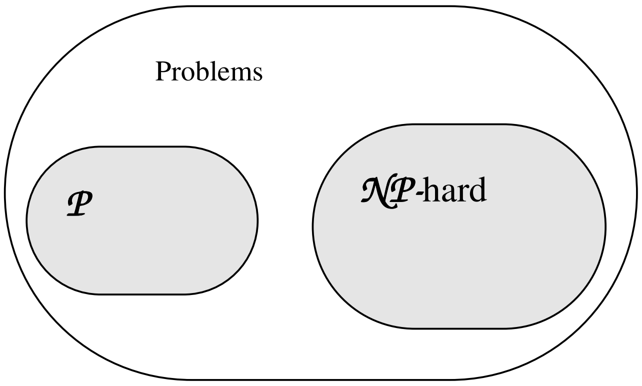
The input size of a typical scheduling problem is bounded by the number of jobs n, the number of machines m and the number of bits to represent the largest integer (the processing time, the due date etc.) An algorithm is said to be ***polynomial*** or a ***polynomial-time*** algorithm, if its running time is bounded by a polynomial in input size. For scheduling problems, typical values of the running time are e.g., $O(n^2)$ and $O(nm)$. Here we write $O(n^2)$ using the big O-symbol to stress that the number of elementary computations of the algorithm grows at the same rate as the function Cn^2, where C is a constant.

Many scheduling algorithms contain the sorting of n jobs, which is known to require at most $O(n \log n)$ time.

Polynomial algorithms are sometimes called efficient or simply good. The class of all polynomially solvable problems is called class $\mathcal{P}$.

Another class of optimisation problems is known as $\mathcal{NP}$-hard problems. For such problems, no polynomial-time algorithms are known and it is generally believed that these problems cannot be solved in polynomial time.

The following is widely accepted: If a problem is $\mathcal{NP}$-hard ($\mathcal{NP}$-complete) it is unlikely that it admits a polynomial-time algorithm, and should be treated by other methods.



References

M.R. Garey and D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", Freeman, San Francisco, 1979.

C.H. Papadimitriou, "Computational Complexity", Addison-Wesley, 1994.

2. Approaches to Scheduling Problems

Given a scheduling problem we first need to determine its complexity status. This is done either by designing a polynomial-time algorithm for its solution or by proving that the problem is $\mathcal{NP}$-hard. There is a web site http://www.mathematik.uni-osnabrueck.de/research/OR/class/ maintained by P.Brucker, S. Knust (University of Osnabrück, Germany) that contains a fairly complete complexity classification of scheduling problems. You can check the complexity status of the problem using LiSA by choosing *<Extras<Problem Classification* from the main menu of the software.

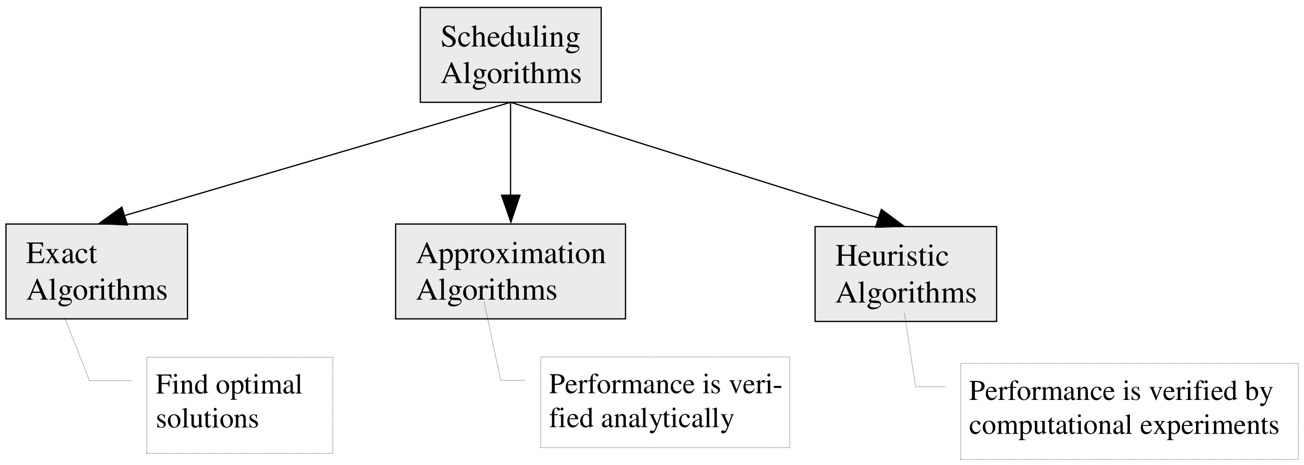
Many scheduling problems turn out to be NP-hard. As a consequence, it is unlikely that those problems can be solved efficiently, i.e. by polynomial time algorithms.

Once we know that our problem is $\mathcal{NP}$-hard, we need to determine whether we want its exact solution or may be happy with an approximate solution. It is unlikely that an exact solution can be found by a polynomial-time algorithm. An exact solution can be found by various methods of reduced enumeration, typically by a branch-and-bound algorithm. Alternatively, the problem can be formulated as a mathematical programming problem. In any case, for problems of practical interest only small-size instances can be handled by exact methods. An exact solution to a job shop problem with 10 jobs and 10 machines remained unknown for more than 25 years.

In order to find a “good” solution within an acceptable amount of time, two types of algorithms can be developed:

- approximation algorithms;
- heuristic algorithms.

An algorithm is called an ***approximation*** algorithm if it is possible to establish analytically how close the generated solution is to the optimum (either in the worst-case or on average). The performance of a ***heuristic*** algorithm is usually analysed experimentally, through a number of runs using either generated instances or known benchmark instances. Heuristic algorithms can be very simple but still effective (dispatching rules). Most of modern heuristics are based on various ideas of ***local search*** (neighbourhood search, tabu search, simulated annealing, genetic algorithms, etc.).



Many scheduling problems turn out to be NP-hard. As a consequence, it is unlikely that those problems can be solved efficiently, i.e. by polynomial time algorithms.

In order to find a "good" solution within an acceptable amount of time, two types of algorithms can be developed:

- approximation algorithms;
- heuristic algorithms.

Approximation algorithms produce solutions in ***polynomial time***, but for the price of loss of optimality. The solutions found are ***guaranteed*** to be within a fixed percentage of the actual optimum. Heuristic algorithms produce feasible solutions, which are not guaranteed to be close to optimum.

"Goodness" of approximation algorithm can be estimated by the ratio $\frac{F(S_A)}{F(S_{OPT})}$, where $F(S_A)$ is the value of the objective function for the solution S_A obtained by the approximation algorithm and $F(S_{OPT})$ is the value of the objective function for the optimal solution.

We define a ρ-*approximation algorithm* to be an algorithm that runs in polynomial time and delivers a solution of value at most ρ times the optimum for any instance of the problem, i.e.,

$$\frac{F(S_A)}{F(S_{OPT})} \le \rho.$$

The value of ρ is called a worst-case ratio bound.

3. Approximation algorithm for problem $1|r_j|\Sigma C_j$

Problem $1|r_j|\Sigma C_j$ of scheduling *n* jobs available at their release times r_j is NP-hard if preemption is not allowed and the objective is to minimise total completion time. This means that it is unlikely that there can be developed a fast exact algorithm to find an optimal schedule. Intuitively, the following two heuristics should work well:

- the *earliest completion time* rule: each time select an unscheduled job with the minimum completion time (similar to SPT rule for problem $1|r_j|\Sigma C_j$);
- the *earliest starting time* rule: each time select an unscheduled job with the minimum release date (processing times are ignored).

<u>*These two rules are examples of heuristic algorithms, as for them no accuracy bounds are known.*</u>

We describe an approximation algorithm for $1|r_j|\Sigma C_j$ based on the optimal solution of problem $1|r_j,Pmtn|\Sigma C_j$. The latter problem can be solved efficiently by the *Shortest Remaining Processing Time Rule* (see the handout on single-machine problems).

<u>**Algorithm 'Convert-Preemptive-Schedule'**</u> (modification of SRPT)

1. Solve the preemptive problem $1|r_j,Pmtn|\Sigma C_j$ using SRPT-rule.
2. Sequence the jobs nonpreemptively in the order that they complete in the solution of the preemptive problem $1|r_j,Pmtn|\Sigma C_j$.

For problem $1|r_j|\Sigma C_j$, algorithm 'Convert-Preemptive-Schedule' is a 2-approximation algorithm.

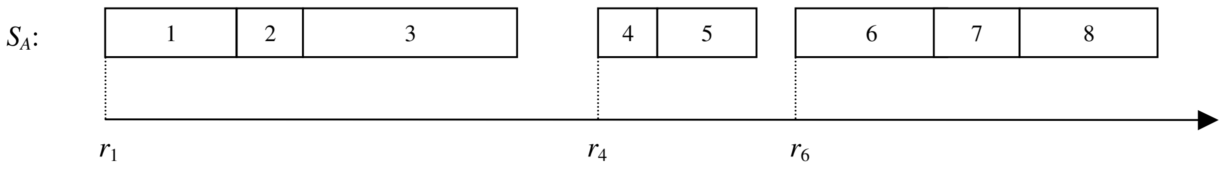
We show that for the schedule S_A constructed by algorithm 'Convert-Preemptive-Schedule' the following inequality holds:

$$\frac{\sum C_j(S_A)}{\sum C_j(S_{OPT})} \le 2,$$

where $C_j(S_A)$ is the completion time of job j in S_A,

$C_j(S_{OPT})$ is the completion time of job j in an optimal schedule S_{OPT}.

Let us renumber the jobs in the order they are completed in schedule S_A. Then the structure of schedule S_A can be illustrated as follows:



The completion time of each job j in schedule S_A is

$$C_j(S_A) = r_u + \sum\nolimits_{k=u}^{j} p_k \ , \qquad (1)$$

where u is the nearest job that precedes job j and starts exactly at its release date r_j (in the example above, u is one of the jobs 1, 4, or 6).

The following two inequalities hold for the ***preemptive*** schedule S_{Pmtn}:

$$\begin{cases} r_u < C_u(S_{Pmtn}) < C_j(S_{Pmtn}), \\ \sum\nolimits_{k=u}^{j} p_k \le C_j(S_{Pmtn}). \end{cases}$$

Substituting these two inequalities in (1), we obtain:

$$C_j(S_A) \le 2C_j(S_{Pmtn}).$$

Finally, since S_{Pmtn} is the optimal preemptive schedule, S_{OPT} is the optimal non-preemptive schedule,

$$\sum C_j(S_{Pmtn}) \le \sum C_j(S_{OPT}).$$

The latter inequality provides the desired ratio guarantee of 2.