Reg. No: 2016/21/D/ST6/02402; Principal Investigator: dr Łukasz Jeż

Throughput Maximization Problems (for general public)

Scheduling problems are ubiquitous: they emerge in production planning, setting up timetables, etc. Hence, they have been rigorously formalized in the early days of computer science, and remain at the core of its fundamental optimization problems to this day. Formalized in terms of jobs and machines, the majority of problems have the objective of minimizing a certain function that depends on the completion times of the jobs. Among those, there are two dominant classes of functions: maximum (e.g., *maximum completion time*, a.k.a. *makespan*) or sum (e.g., *total flow time*). A common theme of such problems is that all jobs are (eventually) completed, but finding the optimal schedule is a challenging task. Specifically, the task is to first partition the jobs between the machines (if there are many) and then to order the jobs assigned to each machine, where typically the first of these two subtasks is (much) harder.

We intend to study another class of problems, where the objective is to maximize the benefit, defined as the number of completed jobs, or more generally, their total weight; these are called *throughput* and *weighted throughput* respectively. The reason why not all jobs can be completed is that they have hard deadlines: completing a job past its deadline brings no benefit, hence such late job may be discarded. Other job characteristics are release (or arrival) times, processing times (or sizes), and weights. Thus, ours are *packing* problems. The *knapsack problem*, i.e., optimizing the total value (which we call weight) of items that can be packed into a knapsack without exceeding its capacity, is a well known very special case: the knapsack corresponds to a single machine, and the items to jobs, all of which share a common release time and deadline, whose difference is the knapsack capacity. We plan to investigate several similar but more general problems: as the knapsack problem and several other are already very well understood, we identify the simplest generalizations that are not.

One example, in which all the jobs have unit sizes, is known as packet scheduling. Note however, that here the jobs (or packets) can have arbitrary release times and deadlines. It turns out that this simply stated problem models quite well the inner workings of network routers, which have to choose which packet should be forwarded next. Clearly, such router operates in a different mode than a "regular" algorithm (or person) trying to pack a knapsack: fine details aside, the crucial difference is that a router has to operate in real-time, and cannot inspect all the packets before choosing one — not only would this be horribly inefficient, it would actually require inspecting (among others) the packets which have not yet arrived at the router! Therefore, we focus on online algorithms, which address the issue of timing and not knowing the future. An online algorithm processes an input sequence piecemeal, where regular atomic bits of the input (e.g., complete information about one job/packet) are interspersed with other bits that request the algorithm to make an action. Such action is an irrevocable decision, based solely on the part of input processed so far. In our scheduling problems the input sequence is ordered by the time, i.e., information about jobs (packets) is revealed at their arrival time, and the algorithm is allowed to make decisions at natural events such as a job arriving or being completed (and some more that we omit here).

We give rigorous performance guarantees by analyzing the worst case, which is typical in algorithm analysis. For online problems, this entails the use of *competitive analysis*, i.e., proving that the algorithm's benefit is at least a certain fraction of the optimum benefit on the same instance, by means of amortized analysis; that guaranteed fraction is called the competitive ratio. We do not explain this in detail, but rather point out that the "the worst case" includes all the possible future events after a given decision of the algorithm. Thus, the guarantees we refer to are very robust: In particular, one may think of the worst case as an input created by a malicious adversary who conjures a future that proves every single decision of the algorithm wrong.

Thinking of the input in such way is very common in online algorithms, and is a foundation of *lower bound* proofs. These are a kind of impossibility results, complementary to algorithm analysis: the goal is showing that no algorithm can ever guarantee a competitive ratio better than certain value. Such results help in understanding if known algorithms can be improved, and to what extent. In particular, without them we can never claim optimality of an algorithm.