3-2018

Working on One Part at a Time is the Best Strategy for Software Production A Proof

Francisco Zapata
University of Texas at El Paso, fazg74@gmail.com

Maliheh Zargaran
University of Texas at El Paso, mzargaran@miners.utep.edu

Vladik Kreinovich
University of Texas at El Paso, vladik@utep.edu

Comments:
Technical Report: UTEP-CS-18-26
To appear in *Proceedings of the 11th International Workshop on Constraint Programming and Decision Making CoProd'2018*, Tokyo, Japan, September 10, 2018

Working on One Part at a Time is the Best Strategy for Software Production: A Proof

Francisco Zapata[1], Maliheh Zargaran[2], and Vladik Kreinovich[2]

[1] Department of Industrial, Manufacturing, and Systems Engineering
University of Texas at El Paso, El Paso, TX 79968, USA
fazg74@gmail.com
[2] Department of Computer Science
University of Texas at El Paso, El Paso, TX 79968, USA
mzargaran@miners.utep.edu, vladik@utep.edu

Abstract. When a company works on a large software project, it can often start recouping its investments by selling intermediate products with partial functionality. With this possibility in mind, it is important to schedule work on different software parts so as to maximize the profit. These exist several algorithms for solving the corresponding optimization problem, and in all the resulting plans, at each moment of time, we work on one part of software at a time. In this paper, we prove that this one-part-at-a-time property holds for all optimal plans.

1 Formulation of the Problem

It is possible to start earning money before the whole software package is released. When a company designs a software package, usually, it does not have to wait until the whole package is fully functional to profit from sales: the company can often start earning money once some useful features are implemented.

As an example, let us consider a company that designs a package for all kinds of numerical computations, including solving systems of equations, optimization, etc. Instead of waiting until all the parts of the software are ready, the company can first release – and start selling – the parts that solve systems of equations. Thus, the company can start earning money before the whole package is ready for use.

This possibility is critical. Indeed, for large software packages, full design can take years. So, the possibility to recoup at least some of the original investment earlier makes such long-term projects more acceptable to managers and shareholders – and thus, makes these projects more probable to be approved; see, e.g., [1].

In view of this possibility, what is the optimal release schedule for different parts? How can we best take into account the possibility to earn money before the package is fully ready, when only some parts of it are ready? What is the optimal schedule for releasing different parts? In what order should we work on them?

Let us formulate this problem in precise terms. The software projects consist of several parts. Some of these parts depend on others, in the sense that we cannot design one part until the other part is ready.

For example, many numerical techniques for solving systems of *nonlinear* equations use linearization, and thus, solve systems of *linear* equations at different stages. So:

- in order to design a part for solving systems of nonlinear equations,
- we need to have available a part for solving systems of linear equations.

This dependency relation makes the set of all parts into a partially ordered set, i.e., in other words, into a directed acyclic graph.

For each part i, we know the overall effort e_i (e.g., in man-hours) that is needed to design this part (by utilizing, if needed, all the parts on which it depends). We also know the profit p_i that we can start earning once this part is released – by adding this part's functionality to whatever we were selling before.

So, if we release part i at time t_i, then by some future moment of time T, selling this part will bring us the profit of $p_i \cdot (T - t_i)$.

The question is: how can we organize our work on different parts so as to maximize the resulting overall profit.

What is known. In [1], several semi-heuristic strategies are described that lead to optimal (or at least close-to-optimal) release schedules.

Interestingly:

- while it is, in principle, possible for the company to work on several parts at a time,
- in all known optimal schedules, the design is performed one part at a time.

What we do in this paper. In this paper, we show that the above empirical fact is not a coincidence: we will actually prove that in the optimal schedule, we *should* always work on one part at a time.

2 Main Result

Exact formulation of the main result that we prove in this paper. What we will prove is that *for each planning problem, there is an optimal schedule in which we always work on one part at a time.*

Discussion. This result does not necessarily means that in *all* optimal scheduled, we work on one part at a time. Let us give a simple example.

Suppose that the software consists of three parts:

- The first two parts are independent and require the same time $t_1 = t_2$.
- The third part depend on the first two parts.

Suppose also that there is no market need for Part 1 or for Part 2, only for the final Part 3. In other words, we assume that $p_1 = p_2 = 0$.

In this case, it is reasonable to conclude that the following schedule is optimal:

- first, we work on Part 1; this take time t_1;
- then, we concentrate all or efforts on Part 2; this also takes time $t_1 = t_2$;
- after this, we work on Part 3; this takes time t_3.

So, by time $2t_1 + t_3$, we get the product that we can start selling.

Alternatively, we can use a different schedule:

- first, we split the team into two equal sub-teams, with one sub-team working on Part 1, and the other sub-team working on Part 2; since designing each part takes time $t_1 = t_2$ for the whole team, it will take twice longer for the twice-smaller sub-teams;
- after the time $2t_1$, both Part 1 and Part 2 are ready, so we can start working on Part 3.

At the end, after time $2t_1 + t_3$, we will get the ready-to-sell product.

Comment. In this example, at least one of the parts does not bring any profit, i.e., has $p_i = 0$. We will see that if each part can bring some profit, i.e., if $p_i > 0$ for all i, then such examples are not possible, and in *all* optimal schedules, we work on one part at a time.

3 Proof

Transformation and its properties. Let us start with an optimal schedule in which at some moment of time, we work on several parts at the same time.

Let t_f be the first moment of time with this property. If any part which is ready by this moment is not yet released in the original optimal plan, we can release it right away and thus provide the additional income from selling this part. Thus, without losing generality, we can assume that:

- all the parts which are released after moment t_f
- are not yet fully ready for release at the moment t_f.

Let t_r be the first moment of time after t_f at which one of the parts is being released, and let j be the number of the part that is released at moment t_r.

Then, we can modify the original schedule as follows.

- Right after the moment t_f, we concentrate all our efforts on this Part j – and all the efforts aimed at other parts are performed after that.
- By the time t_r, we spend the exact amount of efforts on all the parts as before.
- However, since in the original plan, around moment t_f, we were also working on some other parts, this concentration means that we can now release Part j earlier – while preserving the release dates for all other parts.

Thus, in comparison with the original plan, we can earn more (or at least same amount of) profit, since we started selling Part j earlier.

Two possibilities. The profit p_j is always non-negative. So, we have two options: $p_j > 0$ and $p_j = 0$. Let us consider these two options one by one.

Case when $p_j > 0$. If $p_j > 0$, we indeed get more profit – which contradicts to our assumption that the original plan was optimal. Thus, when $p_i > 0$ for all parts, in the optimal plan, we cannot have moments of time at which we work on several parts at a time – which is exactly what we are trying to prove.

Case when $p_j = 0$. If $p_j = 0$, then we do not get any contradiction, but we get a new optimal plan in which:

- either we always work on one pat at a time
- or the first moment of time at which we work on several parts at a time moves further, to some moment $t'_f > t_f$ – since in the vicinity of the moment t_f, we now work on only one part.

In the new plan, we have at least one fewer part on which we work simultaneously with others. Please note that in the new plan, when we work on Part j, we always work *only* on this part. Indeed:

- This has been true – and remains true – before the moment t_f, since t_f is the first moment at which we work on several parts at a time.
- We have also designed a new plan in such a way that this is also true for all moments $t \geq t_f$.

If the resulting plan is not the desired one, we can apply similar transformations. If in the new plan, we still have moments of time when we simultaneously work on several parts, then, by applying the same transformation to the new first-such-moment t'_f, we get yet another optimal plan for which:

- either there is no time when we several parts at a time,
- or the first moment t''_f when this happens is even further away.

Let us prove that this process converges to the desired plan. At each such step, we delete at least one part from the list of parts on which we work simultaneously with others. Thus, after at most as many steps as there are parts, we will get an optimal plans in which:

- no such parts remain and thus
- we always work on one part at a time.

This final optimal plan is exactly the plan whose existence we wanted to prove. Thus, our main result is proven.

Acknowledgments

This work was supported in part by the US National Science Foundation grant HRD-1242122.

References

1. M. Denne and J. Cleland-Huang, *Software by Numbers: Low-Risk, High-Return Development*, Prentice Hall, Upper Saddle River, New Jersey, 2004.