

Time-versus-Capacity Compromises in Project Scheduling¹

Claude Le Pape, Philippe Couronné, Didier Vergamini
and Vincent Gosselin

ILOG S.A.
2 Avenue Galliéni BP 85
F-94253 Gentilly Cedex

Abstract: In most common formulations of project scheduling problems, the objective of scheduling is defined as the minimization of a temporal criterion such as the overall project duration. In actuality, the problem consists not only in minimizing the makespan of the project, but also in avoiding peaks in the utilization of resources over time. Constraint-based algorithms for dealing with this problem are proposed and discussed, as well as their implementation in ILOG SOLVER, a generic C++ object-oriented constraint programming tool [Puget 92] [Puget 94], augmented with ILOG SCHEDULE, a library designed to simplify the representation and the resolution of scheduling and resource allocation problems [Le Pape 93] [Le Pape 94]. An algorithm for determining all the Pareto-optimal trade-offs between makespan and peak capacity is described. Experimental results are provided in the particular case of a ship loading problem presented by Aggoun and Beldiceanu as a benchmark for the CHIP cumulative constraint [Aggoun 92]. In particular, it is shown that the "optimal finishing dates" provided in [Aggoun 92] are not the optimal makespans.

1. Introduction

Project scheduling problems are often solved without considering resource capacity limitations or, in the best of cases, under the assumption that the capacity of each resource (e.g., the number of workers with a given skill) is bounded by a given number. The resulting "standard" formulation of the project scheduling problem is consequently very poor, as the actual problem consists not only in determining the minimal makespan of the project, but also in avoiding peaks in the utilization of resources over time.

This suggests the extension of the most common definition of the project scheduling problem:

- "Given a set of m resources $\{R_1 \dots R_m\}$ with given capacities c_1, \dots, c_m , a set of n activities $\{A_1 \dots A_n\}$ with given durations d_1, \dots, d_n , a set of temporal constraints imposing precedences and delays between start and/or end times of activities, and a set of resource requirements specifying for each i and j , $1 \leq i \leq m$, $1 \leq j \leq n$, the quantity q_{ij} of resource R_i needed for the execution of activity A_j ,
- assign start and end times to the activities, so as to satisfy the constraints and minimize the duration of the overall project"

to a larger class of problems, accounted for by the following, more general, definition:

¹ In: Proceedings of the Thirteenth Workshop of the UK Planning Special Interest Group, Strathclyde, United Kingdom, 1994.

- "Given a set of m resources $\{R_1 \dots R_m\}$ with given maximal capacities $c_1^{\max}, \dots, c_m^{\max}$, a set of n activities $\{A_1 \dots A_n\}$ with given minimal and maximal durations $d_1^{\min}, d_1^{\max}, \dots, d_n^{\min}, d_n^{\max}$, a set of temporal constraints imposing precedences and delays between start and/or end times of activities, and a set of resource requirements specifying for each i, j and d , $1 \leq i \leq m$, $1 \leq j \leq n$, $d_1^{\min} \leq d \leq d_1^{\max}$, the quantity q_{ij}^d of resource R_i needed for executing activity A_j in d units of time,
- assign start and end times to the activities, so as to satisfy the constraints and realize the best trade-off between the duration of the overall project and the peak capacities of the resources".

The aim of this paper is to report on a series of efforts made to solve such problems, using ILOG SOLVER, a generic C++ object-oriented constraint programming tool [Puget 92] [Puget 94], and ILOG SCHEDULE, a library designed to simplify the representation and the resolution of scheduling and resource allocation problems [Le Pape 93] [Le Pape 94]. Section 2 presents the basic characteristics of SOLVER and SCHEDULE. Sections 3 and 4 present SOLVER and SCHEDULE-based algorithms aimed at the resolution of project scheduling problems: section 3 focuses on the case of activities with fixed durations d_i and fixed resource demands q_{ij} ; section 4 discusses the case of activities with variable durations and variable resource demands. Experimental results are presented in section 5.

2. Constraint-Based Programming with ILOG SOLVER and ILOG SCHEDULE

Generally, a *constraint* denotes a relation, in the mathematical sense of the term, between the possible values of variables. For example, if n is a variable which denotes an integer, $n < 10$ is a constraint on the integer n . A constraint satisfaction problem consists of a number of variables and a number of constraints on the values of these variables. Solving a constraint satisfaction problem consists in finding a value for each variable so as to satisfy all the stated constraints. The resolution of a constraint satisfaction problem generally requires the use of a tree search procedure.

Constraint-based programming consists in using the constraints to reduce the computational effort (the exploration of the search tree) needed to solve the problem. Constraints are used not only to test the validity of a solution, as in standard programming languages, but also in a constructive mode to deduce new constraints and rapidly detect inconsistencies. For example, from $x < y$ and $x > 8$, we deduce, if x and y denote integers, that the value of y is at least 10. If later we add the constraint $y \leq 9$, a contradiction is immediately detected. Without propagation, the " $y \leq 9$ " test could not be performed before the instantiation of y : no contradiction would be detected at this stage of the problem-solving process.

So far, three types of constraint-based programming models can be distinguished:

- In the first type of model, generally limited to discrete problems, a constraint satisfaction problem is described as a graph. The nodes of the graph represent the problem variables, to which finite value domains are associated, and the edges the constraints (cf. [Kumar 92]).
- The second type of model stems from logic programming, where constraint solving can be viewed as an extension of unification [Ait-Kaci 91] [Colmerauer 90] [Havens 90] [Jaffar 87] [Van Hentenryck 89].

- Finally, a third type of model distinguishes the concept of a generic constraint (a relation over a chosen interpretation domain) and the concept of a constraint applied to specific variables of the problem to solve [Caseau 91] [Le Pape 88] [Liu 92] [Puget 92] [Siskind 93], without enforcing a problem-solving process based on the unification concept. The main advantage of this type of model is that it allows the user to define very precisely the constraints of the problem to solve on the one hand, and to implement the most appropriate problem-solving strategy on the other hand. In addition, it permits the close integration of constraint-based methods with an object-oriented programming environment [Puget 92].

ILOG SOLVER [Puget 92] [Puget 94] is a C++ library of object classes, functions and control structures, based on the third type of model. It includes a constraint propagation engine, predefined classes of variables and constraints, functions allowing the definition of new classes of constraints, and a full set of primitives for the development of nondeterministic search algorithms.

ILOG SCHEDULE is a SOLVER-based library designed to simplify the representation and the resolution of scheduling and resource allocation problems [Le Pape 93] [Le Pape 94]. ILOG SCHEDULE proposes a simple object model for the representation of scheduling and resource allocation problems in terms of "resources" and "activities". The model consists of a series of C++ classes and functions that implement the concepts of "resource" and "activity" in terms of SOLVER variables and constraints.

ILOG SCHEDULE allows the *representation* of scheduling and resource allocation problems in terms of resources and activities. For the *resolution* of the represented problem, the user of SCHEDULE generally relies on SOLVER to implement a nondeterministic search procedure that corresponds to the requirements of the considered application. Also, the user relies on SOLVER when it is necessary to extend the problem representation to account for domain-specific constraints such as "the duration of the cooling activity equals twice the duration of the preceding heating activity" or "the preparation of the caper sauce starts when the vegetables are half-cooked". To enable such extensions, the reference manual of SCHEDULE describes the constraint-based implementation quite thoroughly.

The next sections illustrate how SOLVER and SCHEDULE are used to solve project scheduling problems involving resources.

3. Resolution of Fixed-Demand Fixed-Duration Problems

Let us first focus on the case of activities with fixed durations d_i and fixed resource demands q_{ij} .

3.1. Problem representation

The first step consists in defining the problem. Four types of constraints must be posted:

- To each resource R_i is associated a given "maximal capacity" $c_{i\max}$. The ILOG SCHEDULE `setCapacityMax` function is used to post capacity constraints: the C++ expression `R->setCapacityMax(timeMin, timeMax, c)` specifies that the capacity of the resource R is limited to c over the time period running from `timeMin` to `timeMax` (e.g., over the complete scheduling period).

- To each activity A_j is associated a given duration d_j . The ILOG SCHEDULE `setDuration` function is used to post fixed duration constraints: the C++ expression `A->setDuration(d)` specifies that the duration of activity A is d.
- Temporal constraints impose precedences and delays between start and/or end times of activities. ILOG SCHEDULE functions are available to post such constraints. For example, the C++ expression `A->startsAfterEnd(B, delay)` specifies that the given delay must elapse between the end of activity B and the beginning of activity A.
- Resource requirements specify, for each resource R_i and each activity A_j , the quantity q_{ij} of resource R_i needed for executing A_j . The ILOG SCHEDULE `requires` function is used to post resource requirement constraints: the C++ expression `A->requires(R, q)` specifies that the execution of activity A requires q units of resource R.

Once these constraints are posted, the underlying constraint propagation mechanism guarantees a systematic update of activity earliest and latest start and end times when decisions are made: for each activity A, it is guaranteed that the earliest and latest start and end times of A are consistent with the temporal constraints and with the resource requirements of "scheduled" activities (i.e., activities for which the start and end times have been fixed). This, in turn, guarantees that any attempt to fix the start and end times of all activities will succeed only if the chosen start and end times satisfy all the constraints of the considered problem.

3.2. Minimization of makespan

Once all the constraints are posted, one can use the nondeterministic programming primitives of SOLVER to generate a solution of minimal makespan.

The first thing to do is to define a makespan variable and specify that its value is greater than the end time of any activity in the schedule. For each activity A, the C++ expression `CtGe(makespan, A->getEndVariable())` constrains the makespan variable to be greater than or equal to the end time of A.

The second thing to do is to write a function that (a) generates a solution to the problem and (b) sets the makespan variable to the makespan of the obtained solution. An algorithm of the following type is adopted:

1. Initialize the set of selectable activities to the complete set of activities to schedule.
2. If all the activities have fixed start and end times, set the makespan variable to the makespan of the obtained solution and exit. Otherwise, remove from the set of selectable activities those activities which have fixed start and end times.
3. If the set of selectable activities is not empty, select an activity from the set, create a choice point for the selected activity (to allow backtracking) and schedule the selected activity from its earliest start time to its earliest end time. Then goto step 2.
4. If the set of selectable activities is empty, backtrack to the most recent choice point. (If there is no such choice point, report that there is no problem solution and exit.)
5. Upon backtracking, mark the activity that was scheduled at the considered choice point as not selectable as long as its earliest start and end times have not changed. Then goto step 2.

The correctness of this algorithm is easy to demonstrate. Indeed, the activity A chosen at step 3 must either start at its earliest start time (and consequently end at its earliest end time) or must be postponed to start later. But starting A later makes sense only if other activities prevent A from starting at its earliest start time, in which case the scheduling of these other activities must eventually result (thanks to constraint propagation!) in the update of the earliest start and end times of A. Hence a backtrack from step 4 signifies that all the activities which do not have fixed start and end times have been postponed. This is absurd as in any solution the activity which will start the earliest among those which have been postponed could be set to start at its earliest start time. This is why it is correct to backtrack from step 4 up to the most recent choice point.

To minimize makespan, the above search algorithm merely needs to be encapsulated in a `CtMinimize SOLVER` statement. `CtMinimize` is a `SOLVER` function which receives as its arguments a search algorithm and a criterion to minimize (e.g., the makespan variable). As long as the search algorithm succeeds in generating a solution to the considered problem, `CtMinimize` adds a new constraint stating that the value of the criterion must be strictly smaller than the value of the best solution found so far. When the search algorithm fails (reports that there is no better solution), it is known that the best solution found so far is optimal.

Using `SOLVER`, the algorithm above is implemented in less than one page of C++ code, including the specification of heuristics to prioritize activities at step 3 (i.e., to decide which activity to select and schedule first). Depending on the problem and on the heuristics used, additional ways of decomposing the problem and cutting the search tree may also be considered. For example, if the problem includes an activity A such that any other activity B

- either is constrained to execute before activity A (i.e., a temporal constraint states that B precedes A),
- or is constrained to execute after activity A (i.e., a temporal constraint states that A precedes B),

then the overall problem may be decomposed in two subproblems, one for the activities which are constrained to precede A, the other for the activities which are constrained to follow A. Needless to say, taking advantage of such an opportunity of decomposing the problem significantly reduces the amount of search required for the generation of the optimal solution. There is, in this respect, much work to do in the domain of project scheduling under resource constraints.

3.3. Generation of Pareto-optimal solutions

Given a problem and two optimization criteria C_1 and C_2 , a solution S to the problem is said "Pareto-optimal" if and only if for any other solution S' , either $C_1(S)$ is smaller than or equal to $C_1(S')$ or $C_2(S)$ is smaller than or equal to $C_2(S')$. In other terms, a solution is Pareto-optimal if and only if there is no way to improve it with respect to one criterion without deteriorating it in terms of the other criterion.

In the context of project scheduling, it is generally interesting to determine the Pareto-optimal trade-offs between the project makespan and the capacity of a particular resource. This lead us to the design of the following algorithm:

1. Generate a solution which minimizes the first criterion C_1 . (If there is no solution at all, exit.) Let V_1 be the optimal value for C_1 .
2. Constrain C_1 to be smaller than or equal to V_1 (posting a new constraint) and generate a solution which minimizes the second criterion C_2 . Let V_2 be the optimal value for C_2 . The solution found is a Pareto-optimal solution to the problem.
3. Remove the constraint stating that C_1 equals V_1 . Replace it by a constraint stating that C_2 must be strictly smaller than V_2 . Goto step 1.

It is easy to verify that any solution S generated in step 2 is Pareto-optimal. Indeed, assume a solution S' is such that $C_1(S') \leq C_1(S)$ and $C_2(S') \leq C_2(S)$. This implies that S' is a solution to the problem considered at step 1, so $V_1 \leq C_1(S') \leq C_1(S) \leq V_1$. Hence $C_1(S') = C_1(S)$. This implies that S' is also a solution to the problem considered at step 2, so $V_2 \leq C_2(S') \leq C_2(S) = V_2$. Hence $C_2(S') = C_2(S)$. Hence there exists no solution S' such that $C_1(S') \leq C_1(S)$ and $C_2(S') \leq C_2(S)$ and either $C_1(S') < C_1(S)$ or $C_2(S') < C_2(S)$. Hence S is Pareto-optimal.

It is also easy to verify that the algorithm determines all Pareto-optimal trade-offs, i.e. that for every Pareto-optimal solution S , the algorithm generates either S or a solution equivalent to S with respect to both C_1 and C_2 . Indeed, assume $(C_1(S) \ C_2(S))$ is a Pareto-optimal trade-off. Two cases can occur:

- First case: $(C_1(S) \ C_2(S))$ is the Pareto-optimal trade-off for which $C_2(S)$ is maximal. Then $C_1(S)$ is minimal and the solution S' generated at the first execution of step 2 satisfies $C_1(S') = C_1(S)$ and $C_2(S') = C_2(S)$.
- Second case: $(C_1(S) \ C_2(S))$ is not the Pareto-optimal trade-off for which $C_2(S)$ is maximal. In this case, let $(C_1(S') \ C_2(S'))$ be a Pareto-optimal trade-off which satisfies the following conditions: (a) $(C_1(S') \ C_2(S'))$ is found by the algorithm; (b) $C_2(S) < C_2(S')$; and (c) $C_2(S')$ is minimal among all the trade-offs that satisfy (a) and (b). At the iteration following the generation of S' , the algorithm will generate a Pareto-optimal solution S'' with $C_2(S'') < C_2(S')$. S'' satisfies (a), hence it cannot satisfy (b). Hence $C_2(S'') \leq C_2(S)$. Hence $C_1(S) \leq C_1(S'')$ as both S and S'' are Pareto-optimal. But then necessarily $C_1(S) = C_1(S'')$, otherwise $C_1(S'')$ would not be a possible value for S'' at step 2. Hence $C_1(S) = C_1(S'')$ and $C_2(S) = C_2(S'')$ and the Pareto-optimal trade-off $(C_1(S) \ C_2(S))$ is found.

Hence the constraint-based algorithm sketched above determines all the Pareto-optimal trade-offs between the two criteria C_1 and C_2 . In the context of project scheduling, C_1 can therefore be defined as the project makespan and C_2 as the maximal capacity of a resource R_i . The definition of the problem must therefore change slightly to link the variable C_2 to the considered resource R_i . This can for example be done by introducing a fake activity which requires a variable amount $c_1^{\max} - C_2$ of R_i from the beginning up to the end of the scheduling period. As a result, only the quantity C_2 remains for the activities actually composing the project.

4. Resolution of Variable-Demand Variable-Duration Problems

In the previous section, we have seen how to generate all the Pareto-optimal solutions of a project scheduling problem with fixed activity durations and fixed resource demands. In actuality, just as the duration of the overall project varies with the total capacity, the duration of an individual activity may vary with the amount of capacity assigned to the activity. In other terms, an activity may require a "variable" amount of capacity, and have a duration which depends on the amount effectively assigned.

In the context of ILOG SCHEDULE, very few changes are necessary to adapt the preceding algorithms to the variable-demand variable-duration case.

- To each activity A_j are associated a given minimal duration d_j^{\min} and a given maximal duration d_j^{\max} .
The ILOG SCHEDULE `setDurationMin` and `setDurationMax` functions are used to post the corresponding minimal duration and maximal duration constraints: the C++ expression `A->setDurationMin(dmin)` specifies that the duration of activity A is at least d_{\min} ; the C++ expression `A->setDurationMax(dmax)` specifies that the duration of activity A is at most d_{\max} .
- In the C++ expression `A->requires(R, q)`, q is allowed to be a constrained variable rather than a constant. Hence nothing changes with respect to the requirement constraints, except the type of q . The domain of q is set to the possible values $q_{i,j,d}$ of the demand and q is related via a constraint to the duration of the activity. For example, the product of the demand q and the duration d may be constrained to be greater than or equal to a given constant C . The C++ statement `CtGe(CtMul(q, d), C)` implements this constraint.
- It is no longer possible, in the algorithm of section 3.2, to make a unique decision for the start and the end time of a selected activity. Indeed, as several values are acceptable for the activity duration, there may be several possible values for the end time of the activity, even when the start time of the activity is fixed. In addition, the proof of the correctness of the algorithm in section 3.2 depended upon the fact that it was always possible to know whether a postponed activity could still be executed from its earliest start time to its earliest end time (i.e., constraint propagation was taking care of modifying these earliest start and end times whenever they became unacceptable values, due to the scheduling of other activities). This property does not make sense any longer as it could be that an activity can start at its earliest start time, or can end at its earliest end time, but cannot both start at its earliest start time and end at its earliest end time. To "fix" the algorithm, it is therefore necessary to set (fix) the duration of the selected activity in step 3 of the algorithm. This is achieved by introducing a new choice point in the search tree, allowing to enumerate the possible durations: the smallest possible duration is tried first; in case of backtracking, the next smallest possible duration is tried; and so on. Of course, this greatly expands the size of the search space!

5. Experimental Results

The algorithms presented above were experimented on two sets of problems. The first set was provided by a client in order to evaluate the contribution of ILOG SOLVER and ILOG SCHEDULE to a given industrial problem. The projects to be scheduled consist of forty to fifty activities requiring workers with two distinct qualifications. The duration of each activity varies with the number of workers assigned to the activity. However, a minimal

and a maximal duration are provided. The problem consists in determining a suitable trade-off between the makespan of the project and the number of workers with each qualification. Through a graphical interface, the user specifies the maximal capacity of each qualification group and runs the algorithm to determine the corresponding optimal makespan. Then the user can change each of the maximal capacities and run the algorithm again. The average run requires less than a second of computational time (on a SUN-4 workstation). It is also possible to specify that a number of workers do possess both qualifications, in order to study the possible effects of additional worker training on the project.

The ship loading problem presented by Aggoun and Beldiceanu as a benchmark for the CHIP cumulative constraint [Aggoun 92] was chosen to allow publication of benchmark results. This example consists of 34 activities submitted to the following precedence constraints.

Activity	Successors	Activity	Successors	Activity	Successors	Activity	Successors
1	2, 4	11	13	21	22	31	28
2	3	12	13	22	23	32	33
3	5, 7	13	15, 16	23	24	33	34
4	5	14	15	24	25	34	
5	6	15	18	25	26, 30, 31, 32		
6	8	16	17	26	27		
7	8	17	18	27	28		
8	9	18	19, 20, 21	28	29		
9	10, 14	19	23	29			
10	11, 12	20	23	30	28		

One of us (the authors) focused on the fixed-demand fixed-duration example given in [Aggoun 92]. A unique resource is considered, the capacity of which is set to 8. The duration of each activity and the capacity required by each activity are set to the following values.

Act.	Dur.	Cap.	Act.	Dur.	Cap.	Act.	Dur.	Cap.	Act.	Dur.	Cap.
1	3	4	11	3	4	21	1	4	31	2	3
2	4	4	12	2	5	22	2	4	32	1	3
3	4	3	13	1	4	23	4	7	33	2	3
4	6	4	14	5	3	24	5	8	34	2	3
5	5	5	15	2	3	25	2	8			
6	2	5	16	3	3	26	1	3			
7	3	4	17	2	6	27	1	3			
8	4	3	18	2	7	28	2	6			
9	3	4	19	1	4	29	1	8			
10	2	8	20	1	4	30	3	3			

The optimal solution and the proof of optimality are obtained in 0.2 seconds (on a SUN-4 workstation). Figure 1 displays the optimal solution. This solution is better than the solution presented as "optimal" in [Aggoun 92] (cf. figure 2).

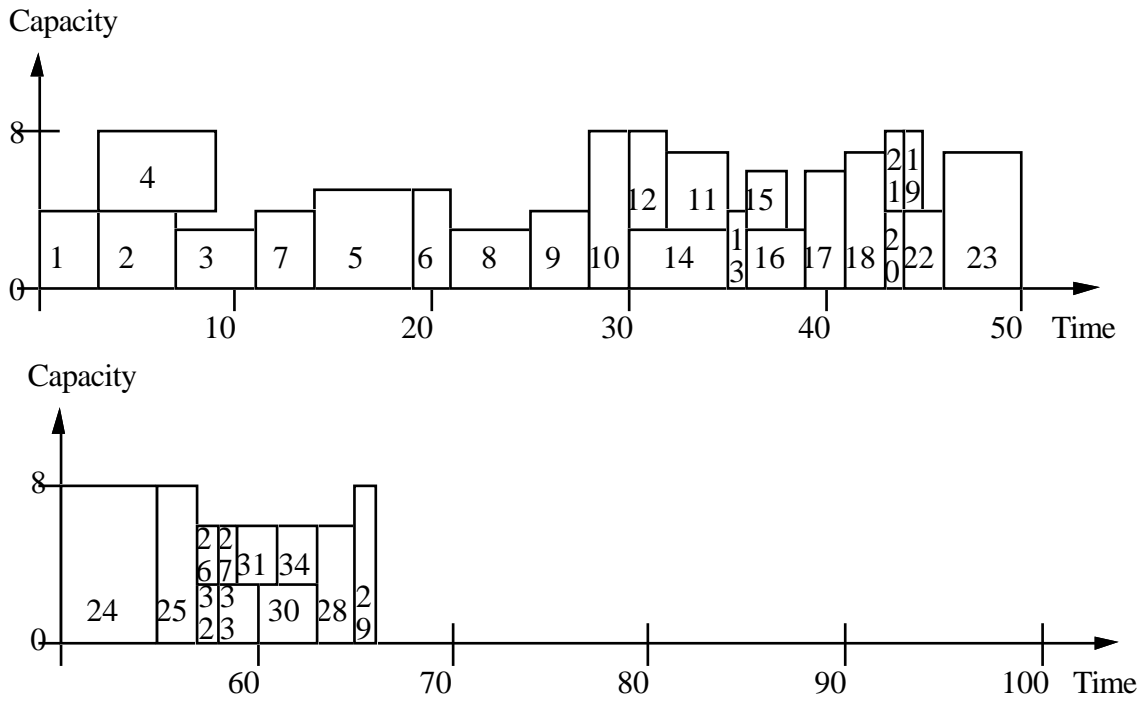


Figure 1: An optimal solution to the ship loading problem

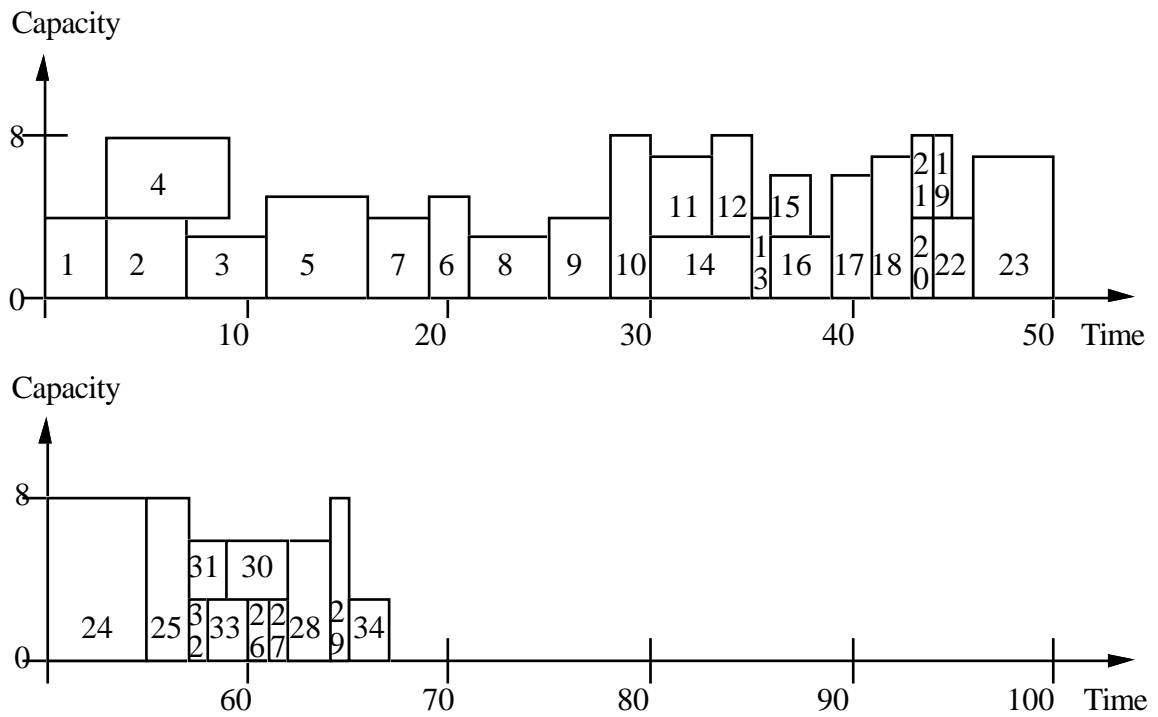


Figure 2: The “optimal” solution of [Aggoun & Beldiceanu 92]

Another of us developed the algorithm allowing the determination of all Pareto-optimal trade-offs and tested it on the fixed-demand fixed-duration example above. Three Pareto-optimal solutions and the proof that there are only three Pareto-optimal trade-offs are obtained in 0.4 seconds. The three Pareto-optimal solutions necessitate respectively:

- 66 units of time for maximal capacity 8;
- 59 units of time for maximal capacity 9;
- 58 units of time for maximal capacity 11.

These results illustrate the interest of intermediary compromises. Indeed, a project manager may be willing to add one unit of capacity in order to save seven units of time, but probably will not want to add two additional units of capacity, in order to save one unit of time.

The two other authors developed the algorithm allowing the consideration of variable-demand variable-duration examples. Following Aggoun and Beldiceanu, they tested the algorithm on the example above modified to allow the duration and the required capacity to vary, but with the product of duration and demand fixed to a constant value. For example, the possible (duration demand) pairs for the first activity are (1 12), (2 6), (3 4), (4 3), (6 2) and (12 1), the possible (duration demand) pairs for the second activity are (1 16), (2 8), (4 4), (8 2) and (16 1), etc. The following table displays the results.

Capacity	Makespan	CPU Time	Backtracks	Capacity	Makespan	CPU Time	Backtracks
1	393	0.12	0	21	30	0.20	6
2	203	0.22	32	22	29	0.15	7
3	168	0.86	370	23	29	0.16	6
4	121	0.74	311	24	29	0.13	4
5	90	0.73	271	25	25	0.17	18
6	81	0.51	152	26	25	0.16	11
7	67	0.70	326	27	25	0.16	9
8	55	0.35	78	28	23	0.15	8
9	52	0.38	82	29	23	0.16	8
10	49	0.58	227	30	23	0.16	8
11	47	0.27	34	31	23	0.15	6
12	42	0.36	64	32	23	0.15	5
13	42	0.37	99	33	23	0.15	5
14	37	0.37	90	34	23	0.15	5
15	35	0.26	53	35	23	0.15	5
16	33	0.26	69	36	23	0.16	3
17	33	0.28	54	37	23	0.13	0
18	32	0.26	35	38	23	0.13	0
19	32	0.22	10	39	23	0.13	0
20	30	0.19	9	40	22	0.12	0

Figure 3 displays the optimal solution obtained under these constraints for maximal capacity 8. Figure 4 displays the optimal solution for maximal capacity 5. Computational times are on the average similar to those of [Aggoun 92] but much more stable, with a standard deviation of 0.19 seconds (1.17 seconds in [Aggoun 92]). The reported makespans are always equal to those of [Aggoun 92] minus 1. We have been unable to determine whether this corresponds to a different convention for the start time of the project or to sub-optimality of the [Aggoun 92] results (as is the case in figure 2).

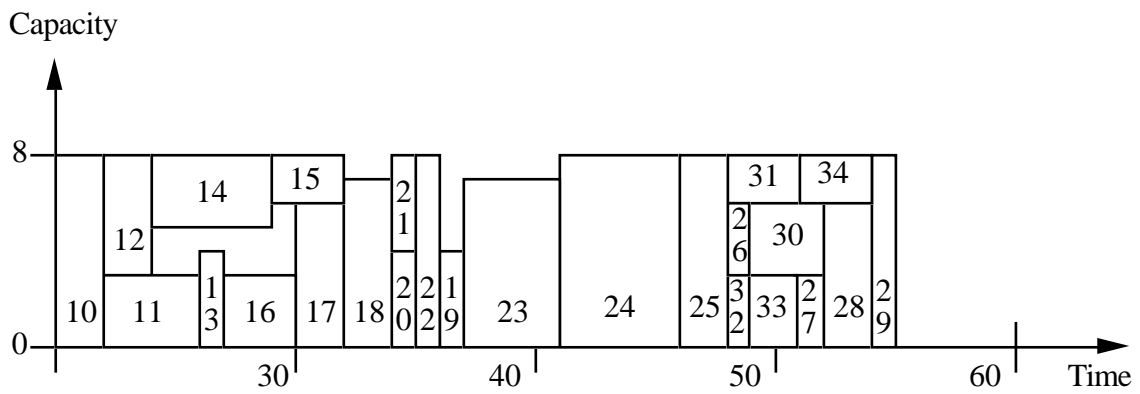
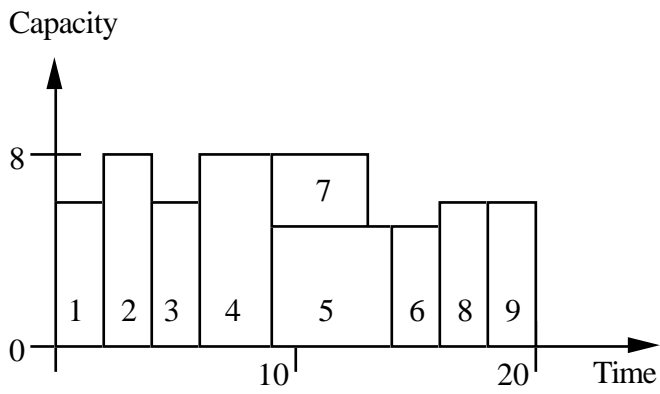


Figure 3: An optimal solution to the “variable demand” problem (capacity = 8)

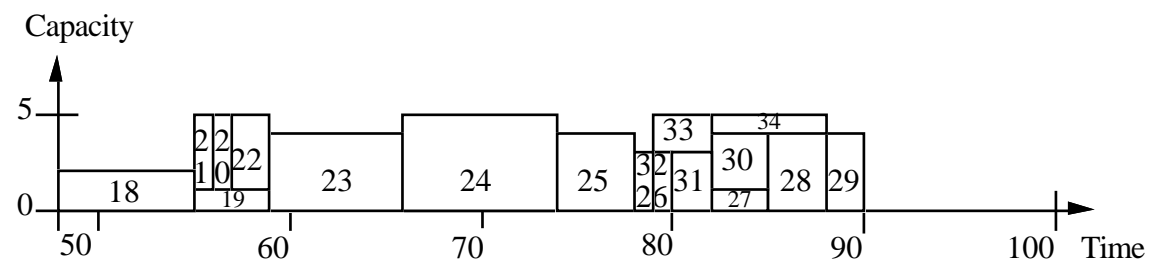
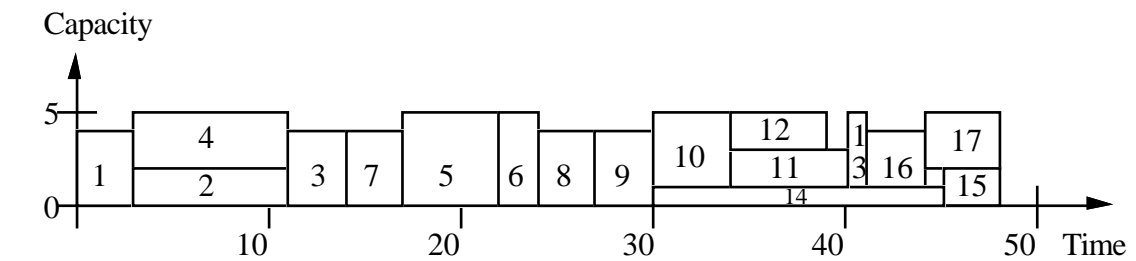


Figure 4: An optimal solution to the “variable demand” problem (capacity = 5)

Finally, the algorithm allowing the determination of Pareto-optimal solutions was applied to the variable-demand variable-duration example. It computes the 21 Pareto-optimal trade-offs and proves that there are only 21 Pareto-optimal trade-offs in 23 seconds.

6. Conclusion

Most common formulations of project scheduling problems assume that the scheduling objective is to minimize a temporal criterion such as the overall project duration. In actuality, it is often interesting to determine the possible trade-offs between time and capacity requirements. In particular, a constraint satisfaction algorithm allowing the determination of Pareto-optimal solutions to the time-versus-capacity problem can be of high interest. Such an algorithm is easily implemented on top of ILOG SOLVER and ILOG SCHEDULE. In this respect, constraint programming tools may play an important role toward the development of new project scheduling systems, ensuring a better coverage of the requirements of project management activities.

7. References

- [Aggoun 92]
Abderrahmane Aggoun and Nicolas Beldiceanu.
Extending CHIP in Order to Solve Complex Scheduling and Placement Problems.
Premières journées francophones sur la programmation en logique, Lille, France, 1992.
- [Ait-Kaci 91]
Hassan Ait-Kaci and Andreas Podelski.
Is there a Meaning to LIFE?
PRL Technical Report, Digital Equipment Corporation, 1991.
- [Caseau 91]
Yves Caseau.
Abstract Interpretation of Constraints on Order-Sorted Domains.
Bellcore Technical Memorandum, 1991.
- [Colmerauer 90]
Alain Colmerauer.
An Introduction to Prolog III.
Communications of the ACM, **33**(7):69-90, 1990.
- [Havens 90]
William S. Havens, Susan Sidebottom, Greg Sidebottom, John Jones, Miron Cuperman and Rod Davison.
Echidna Constraint Reasoning System: Next-generation Expert System Technology.
Technical Report, Simon Fraser University, 1990.
- [Jaffar 87]
Joxan Jaffar and Jean-Louis Lassez.
Constraint Logic Programming.
Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, West Germany, 1987.
- [Kumar 92]
Vipin Kumar.
Algorithms for Constraint Satisfaction Problems: A Survey.
AI Magazine, **13**(1):32-44, 1992.

[Le Pape 88]

Claude Le Pape.

Des systèmes d'ordonnancement flexibles et opportunistes.

PhD Thesis, University Paris XI, 1988 (in French).

[Le Pape 93]

Claude Le Pape.

The Cost of Genericity: Experiments with Constraint-Based Representations of Time Tables.

Proceedings of the Sixth International Conference on Software Engineering and its Applications, Paris La Défense, France, 1993.

[Le Pape 94]

Claude Le Pape.

Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems.

Intelligent Systems Engineering, **3**(2):55-66, 1994.

[Liu 92]

Bing Liu and Yuen-Wah Ku.

ConstraintLisp: An Object-Oriented Constraint Programming Language.

ACM SIGPLAN Notices, **27**(11):17-26, 1992.

[Puget 92]

Jean-François Puget.

Programmation par contraintes orientée objet.

Douzièmes journées internationales sur les systèmes experts et leurs applications, Avignon, France, 1992 (in French).

[Puget 94]

Jean-François Puget.

A C++ Implementation of CLP.

Technical Report, Ilog S.A., 1994.

[Siskind 93]

Jeffrey M. Siskind and David A. McAllester.

Nondeterministic Lisp as a Substrate for Constraint Logic Programming.

Proceedings of the Eleventh National Conference on Artificial Intelligence, Washington, District of Columbia, 1993.

[Van Hentenryck 89]

Pascal Van Hentenryck.

Constraint Satisfaction in Logic Programming.

MIT Press, 1989.