Production Systems and Information Engineering
Volume 2 (2004), pp. 107-119

A DETAILED EXAMPLE OF APPLYING CONSTRAINTS ON A LOGISTICAL PROBLEM

ELEMÉR KÁROLY NAGY
Department of Control Engineering and Information Technology,
Faculty of Electrical Engineering and Informatics,
Budapest University of Technology and Economics
eknagy@kempelen.iit.bme.hu

[Received November 2004 and accepted February 2005]

Abstract. This article demonstrates the use of constraints to reduce the algorithmic complexity of industrial problems through a numerically detailed example. To achieve this, the article first presents the basics of algorithmic complexity and constraints as well as the types and uses of constraints. Second, the article presents a logistical example and determines its complexity. Third, constraints are applied on the problem. Fourth, the problem is modeled in a constraint programming environment and the different results received from different constraint sets are evaluated.

Keywords: Algorithmic complexity, constraint programming, logistical problem

1. THE GROWTH OF COMPUTING POWER

In the early days of computing, computing power was so rare and expensive that dozens of scientists had to work on the same computer and they could solve only simple problems. Later, every scientist could have a personal computer and they could do most of their work on it. However, if they had difficult problems they still had to work on the mainframe. At the end of the 20th century, everyone had a personal computer that was almost as powerful as a mainframe, and scientists solved difficult problems on their PCs. When they needed even more computing power, they entered the problem in a cluster of thousands of mainframes. Yet there are problems that cannot be solved even with clusters. These problems have enormous algorithmic complexity and they are believed to take at least a hundred year even if the available computing power is doubled every year.[1]

2. ALGORITHMIC COMPLEXITY

"Algorithmic complexity" is an abstract expression which is used to describe the computation power needed to solve a problem with a given algorithm.[2] It is mostly independent of hardware and implementation details, depending only – at least in theory - on the algorithm, the problem class and the type of resource used.

One such problem class is the N –queen problem, in which we need to find all the solutions of the problem "N queens on an $N*N$ chess table, none of them can hit any of the others in one move"

One such resource is the memory, measured in Memory Units or MUs, which – in our example – denotes the memory needed to store an $N*N$ table with N queens. Another such resource is CPU time, measured in Computational Units or CUs which – in our example – denotes the time needed to check an $N*N$ chessboard if it fulfills the requirements of the N-queen problem. The exact amount of these resources may vary depending on the hardware, the programming language, the data structures, etc. In our example, the memory denoted by "one MU" may vary from about $N*\log_2(N^2)$ bits to $4*N^2$ bytes, namely 6 to 256 bytes for $N=8$.

Using these definitions, we could state that my hypothetical algorithm solves the 4-queen problem with using the maximum of 256 MUs and 128 CUs, solves the 8-queen problem with using the maximum of 2048 MUs and 512 CUs.

To describe the memory and CPU usage of my hypothetical algorithm for all possible Ns, I either need to produce a table that contains every N and the corresponding MUs and CUs or I need to find a formula that generates this table. In practice, it is widely accepted to use the formula of "maximum algorithmic complexity" which is denoted as $O()$ and pronounced as "big ordo" This operator is adopted from Calculus, in which the definition is (1), where C and N are constants and a_n and b_n are sequences:

$$a_n = O(b_n) \Leftarrow \exists C : |a_n| \le C|b_n|, n > N \tag{1}$$

In computer science, "$O(N^3)$ in terms of memory usage" is used to denote that "the given algorithm solves the given problem class with parameter N using the maximum of $C*N^3$ MUs, even in the worst case, where C depends on hardware and implementation but is independent of N" For example, my hypothetical algorithm solves the N-queen problem in $O(N^3)$ MUs and $O(N^2)$ CUs. In this case, $C=4$, but $C=4096$ would still result $O(N^3)$. This is the consequence of the definition and reflects the fact that scalability[1] cannot compensate non-linear algorithmic complexity.

[1] Faster processors, increased storage capacities, multiprocessor systems, clusters, distributed applications, etc.

There is another adopted operator in use, namely $o()$ or "small ordo" with the original Calculus definition of (2) , where c and N are constants and a_n and b_n are sequences:

$$a_n = o(b_n) \Leftarrow \forall c > 0 : |a_n| \leq c|b_n|, n > N \quad (2)$$

This operator is used sometimes incorrectly to denote the "minimum algorithmic complexity" which is slightly different from the exact mathematical meaning[2] This may be tolerable as the minimum algorithmic complexity usually has far less importance in practice than the maximum algorithmic complexity. Therefore, I will use the operator $q()$ to denote the minimum algorithmic complexity, so "$q(N^3)$ in terms of memory usage" is used to denote that "the given algorithm solves the given problem class with parameter N using the minimum of $C * N^3$ MUs, even in the best case, where C depends on hardware and implementation but is independent of N "

There are algorithms/problems where q and O differ. For example, if we are not interested in all solutions of the N -queen problem, just in the first solution, the complexities may be $q(N^2)$ and $O(N^4)$ in terms of memory usage. If q differs from O, it is practical to use both of them when we speak about complexity, but we may refer to O only as it denotes the worst case. In algorithms where q and O are equal, it is sufficient to use O only.

In practice, an English-Mathematician Dictionary [3] contains the following translations: $O(const)$ = "utopian", $O(\log N)$ = "excellent", $O(N)$ = "very good", $O(N * \log N)$ = "decent", $O(N^2)$ = "not so good", $O(N^3)$ = "pretty bad", $O(N^4)$ = "terrible", $O(const^N)$ = "disaster"[3] This sequence of complexities is also referred to as "complexity classes", so an $O(N^2)$ problem is two classes harder than an $O(N)$ problem. Until now, we used to denote q and O to describe the complexity of an algorithm. However, we may use q and O to describe the complexity of a problem, especially if we have an algorithm with known complexity that solves the problem.

[2] If the algorithm uses $4 * N^3$ MUs in every case, it is $q(N^3)$ and $o(N^{3.0001})$ but not $o(N^3)$

[3] $const > 1$

An $O(N * \log N)$ problem denotes a problem that has at least one $O(N * \log N)$ solution. Sorting elements in an unsorted array with N elements is an $O(N * \log N)$ problem in terms of execution time. [4]

In some cases, it is mathematically proven that no faster algorithm exists, these problems are often referred as "proved $O(X)$ problems"

It is generally accepted that an algorithm designed to find a simple solution to a problem is "way faster" than an algorithm designed to find all solutions while algorithms designed to find the best solution are almost identical, at least in the terms of MU and CU, with the ones designed to find all solutions. This is reflected in algorithmic complexity as these three cases define three different problems. The maximum complexities of these problems are often the same and the minimum complexities of these problems almost always differ by at least two classes.

3. INDUSTRIAL LOGISTICS EXAMPLE

As defined in [5], "in an industrial context, logistics means the art and science of obtaining, producing, and distributing material and product in the proper place and in proper quantities" One such logistical problem is the problem of manufacturing cells[4] Manufacturing cells can produce different products but changing the type of product requires time (referred to as retooling). The products and the resources may have different restrictions that are direct consequences of storage capacities, workforce limits, delivery deadlines, etc.

One such example is the following (MU is money unit, TU is time unit): There are 3 identical manufacturing cells (C1, C2, C3). There are 3 resources (R1, R2, R3). There are 4 tools , two of each type (T1/1, T1/2, T2/1, T2/2). T1 tools produce one R3 from two R2s and two R1s in every TU. T2 tools produce three R2s from two R1 in every TU. The storage costs of R1, R2, R3 are 1, 1, 3 MU/TU. Retooling takes 1 TU. R1 and R2 resources can be bought at a price of 5 MU/piece in quantities of 20 while R3 can be sold at a price of 100 MU/piece in quantities of 10. Buying and selling takes 1 MU. At the beginning of the shift, we have 1000 MUs, C1 has T1/1, C2 has T2/2, C3 has T1/2, and we have 100 pieces of R1s, 11 pieces of R2s and 1 pieces of R3. We have to prepare 50 pieces of R3s in the

[4] The problem of manufacturing cells is often treated as scheduling or manufacturing problem when "logistics" is used in a narrower sense. However, when the definition omits production from logistics, then logistics (in a broader sense) becomes a sequence of logistics (in a narrower sense) and production problems. As these problems are not independent, the complexity of the main problem is not reduced but increased.

storage at the end of the 56th TU when it will be removed as ordered by the CEO. We have to achieve the most MUs at the end of the 100th TU. What shall we do?

This example is a simplified real-life example, in which the quality of the solution found has great impact on profit. To find the best solution, a PPS[5] should utilize an algorithm that finds the optimal solution before the shift starts. Let us assume that we have a state-of-the-art computer that can store 10 000 000 states in its memory and can analyze 100 000 states per second. Let us assume that the computer has 24 hours to find the best solution.

4. THE COMPLEXITY OF THE EXAMPLE

The problem of manufacturing cells can be modeled as a single-source multi-destination directed grah search problem. In a graph search problem, there are nodes (also referred as states) and edges (also referred as transitions). In the example, there is a single source node (the beginning of the shift), there are destination nodes (the possible outcomes at the end of the shift) and there are intermediate (mid-shift) nodes. In the example, the nodes are arranged in layers, each layer contains nodes with the same TUs. Also, the edges always connect two adjacent layers and the destination's TU is always higher than the source's TU exactly by one. In a general graph search problem, these restrictions do not apply.

A search algorithm takes the current node and chooses an edge to follow, entering into the next current node. The difference between the search algorithms comes from the difference in the choices they make. If the edges have "distance" or "cost" values and it affects the choices made by the algorithms, we speak of guided search algorithms.

In this example, as each cell can either retool or produce or stay idle, we have 3 possibilities for each cell. We may decide to buy or to not buy 20 pieces of R1s, to buy or not to buy 20 pieces of R2s, to sell or not to sell 10 pieces of R3s, thus we have 2 possibilities for each resource. This means we have to make six choices and thus we have $3*3*3*2*2*2=216$ possibilities in each TU (216 transitions from every state).

The basic unguided algorithm to solve such a problem is the breadth-first search.[6] To find the best solution (the destination state with the most MUs) with the breadth-first search it is necessary to examine all possible states. To find all possible solutions, we would have to check 216^{100} possibilities and we would need to compute for about 10^{221} years with the given computer. It is impossible to

[5] In this context, Production Planning System

do so because finding the most profitable solution(s) for this problem with simple breadth-first search without applying constraints has the complexity of $O(216^N)$ where N is the number of TUs. Guided algorithms like the A* search find one of the best solution faster than unguided search algorithms if certain conditions apply[6] and so if they do not need to check all possible states. In this example, we require that the search algorithm checks all states that are not disqualified by constraints, so the breadth-first search is used for the sake of simplicity.

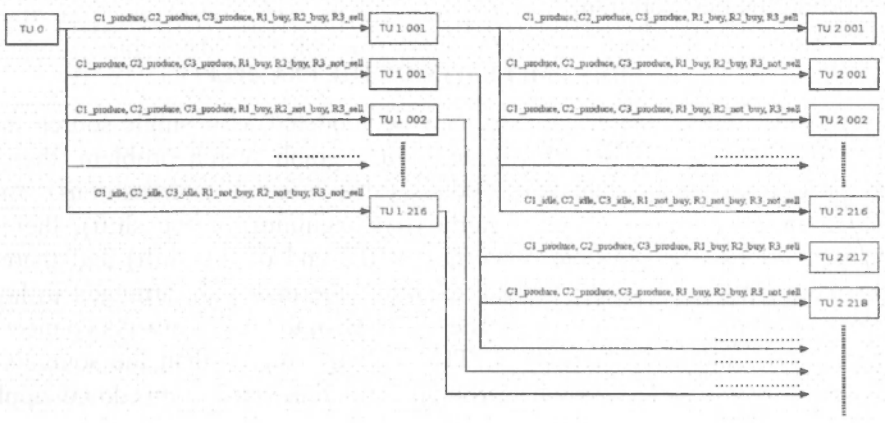


Figure 1: Simple BFS

5. CONSTRAINTS

Constraints are restrictions that reduce the number of possible actions, reducing the number of states and thus they reduce the necessary computing power.[7]

Constraints are either:

- internal problem constraints (we cannot produce R3s if we have no R2 in the storage),
- external problem constraints (we need to have 50 R3s at the end of the 56th TU), or
- algorithm constraints[7] (one algorithm could find out that if we need to have 50 R3s at the 56th TU, then – as we cannot produce more than two R3s in a TU as we have only two T1s – we need at least 48 R3s at the 55th TU, 46 R3s at the 54th TU, , 2 R3s at the 32nd TU).

[6] Detailed discussion is not feasible within the frame of this article

[7] Algorithm constraints include deducted constraints (constraint deducted from other constraints) but also include constraints that are not formally deducted.

A problem can be either – by internal and external constraints:

- over-constrained (there is no solution to the problem);
- under-constrained (the number of solutions is too great); or
- well-constrained.

Algorithm constraints may be added to the problem to reduce the number of transitions from the states as long as they do not remove any of the best solutions of the problem. To make the example solvable, we introduce the following four algorithm constraints:

- Constraint 1: We do not buy 20 R1s if we have 11 or more R1s.
- Constraint 2: We do not buy 20 R2s if there are two T2s equipped or we have 4 or more R2s.
- Constraint 3: We sell 10 R3s if we have more than 9 R3s in the storage and it is past the 56th TU.
- Constraint 4: We sell 10 R3s if we have more than 59 R3s.

These constraints can only be applied to this particular example. When applied, they reduce the 8 possibilities of buying/selling to the average of 3. This results in an enchanted performance as we reduced the number of states to check from 216^{100} to about 81^{100}, which is 10^{43} faster.

According to the definition, the problem still has $O(216^N)$ complexity as O denotes the maximum algorithmic complexity. If we analyzed all states, we would get $O(81^N)$ for the given specific example with the four example constraints added. However, if we change any specific data (the starting amount of R1s, for example) then it might not be $O(81^N)$ while it still would be $O(216^N)$.

6. ADDING AND USING CONSTRAINTS

There are two ways of adding constraints: automatically or manually. Manually added constraints tend to be "stronger" as they reduce the complexity more severely than automatically added constraint, but also have the tendency of disqualifying valid solutions even best solutions due to human error. Automatically added constraints often do not reduce the complexity enough to solve the problem in the given time but they usually speed it up.

If an algorithm tries to check all possible solutions it may use constraints in one of the following ways:

- In every state, the algorithm checks if all the constraints are fulfilled. If any of the constraints is violated, the algorithm steps back (backtrace). For

example, if we are in the 56th TU and we don't have 50 R3s then we go back to the previous state.

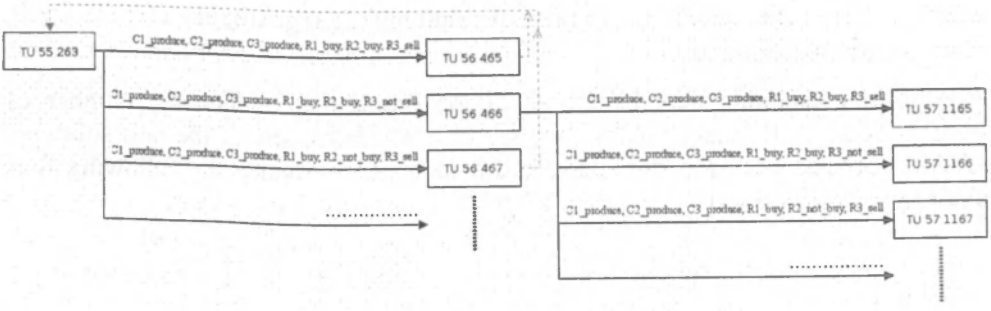


Figure 2: Backtrace

- The algorithm initiates a transition only if it is sure that all constraints will be fulfilled after the transition (forward checking). For example, in the 55th TU we do not sell 10 R3s if we have less than 60 pieces.

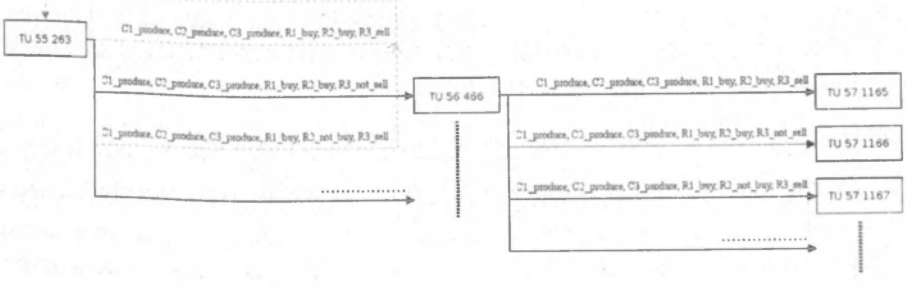


Figure 3: Forward checking

- The algorithm generates new constraints from the available ones and does not initiate a transition that violates any of them (constraint propagation). For example, deducting the need for 2 R3s at the 32nd TU.

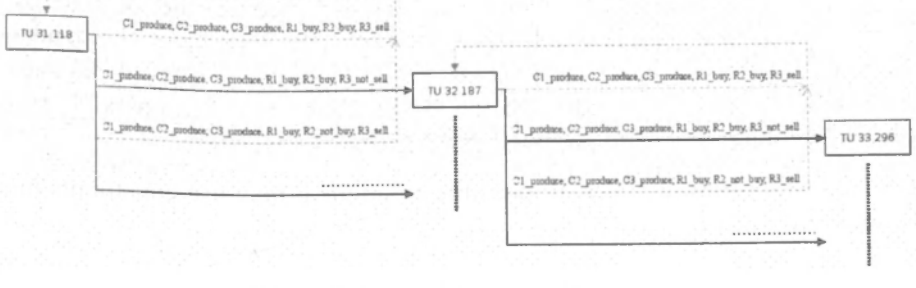
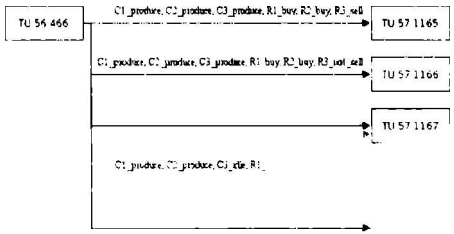


Figure 4: Constraint propagation

7. OTHER METHODS TO REDUCE COMPLEXITY

There are two ways of reducing complexity. Non-destructive reductions are reducing the state space by eliminating unnecessary nodes that can not lead to valid solutions. Destructive reductions reduce the state space by eliminating nodes that may or may not lead to valid solutions, thus possibly reducing the quality of the solution found. Constraints, as long as they are correct, are non-destructive. The other two non-destructive reduction methods are remodeling and equality checking. The destructive reductions include applying policies and changing the granularity.

Remodeling the problem decreases the complexity if the new model is simpler than the original. In our example, we may remodel the three identical cells and the four tools as a single cell with 8 tools, thus decreasing the complexity to $O(64^{100})$ In most cases, there are constraints with the same effect, for example, the ITACF constraint introduced in "Solving the given problem"



Equality checking is based on the fact that there may be identical state-pairs in a system from which any chosen transaction results identical states[8], and it is sufficient to keep only one of them and drop the others as they can not lead to a better solution. If a proper value function is given, it is even possible to find state-groups from which only one state is needed to be preserved. In our example, if two states have the same tool configuration, TUs and resources, it is feasible to store only the one with the highest amount of MUs and drop all other inferior states, using a proper value function.

Policies are constraints that are not formally deducible from other constraints and therefore they might disqualify valid solutions. In our example, one such policy is LRS which is described in "Solving the given problem"

Changing the granularity reduces the state space by reducing the domain of state variables. A linear reduction in the proper variable's domain may reduce the state space exponentially. In our example, by reducing the 100 TUs in a shift to 20 TUs, we reduce the complexity from $O(216^{100})$ to $O(216^{20})$, but we also decrease the quality of the solution. Another example could be splitting the problem into two smaller problems, namely the problem of the first 56 TUs and the problem of the last 44 TUs.

[8] There are many other definitions of „identical" not discussed here.

If the problem is still too complex to be solved by an algorithm in the limited time, there are means to increase the quality of the solution found.

These include random sampling and guided search, which takes many forms from greedy search to A* search. Greedy search always checks the transitions with the most income first, while A* search checks the transition first with the highest value, which is provided by a heuristic function. One such heuristic function may return the total value of stored resources plus the current amount of MUs minus the estimated storage costs until the 100th TU. Random sampling chooses a transition at random, thus – on average – progressing through the state graph evenly.

8. SOLVING THE GIVEN PROBLEM

This particular problem can be solved with a number of tools. One such tool is SCPFW, which is developed by the author and is available at sourceforge.net under GPL. SCPFW is used in the education at BUTE and the given problem can also be, and is solved by SCPFW. The ProductionSytem class in SCPFW is a more abstract problem of manufacturing cells that can be easily customized to implement the given problem. The ProductionSystem has two built-in general constraints and a built-in optimization. The optimization is "Similarity Check" or SC, which drops the states that have less money than their state-pairs, as described in "Other methods to reduce complexity" The first constraint is "Lazy Resource Strategy" or LRS, which disables resource buying as long as the resource can be bought faster than consumed and there is enough resource left. The second is "Identical Tool Action Combinations Filter" or ITACF, which disables transitions that are permutations of other transitions and are identical in their effect. The customized ProductionSystem has a problem-specific constraint, namely "Max R3 Constraint" or MR3, which limits the maximum amount of R3 in the storage. ProductionSystem contains a simple BFS-backtrace algorithm that handles all inner constraints. It tries to execute all transitions from all the N-step states before entering any of the N+1-step states. In case of violation of an inner or outer constraint, a return value is set to false or an exception is generated and the algorithm drops the resulting state. If the transition is executed successfully, the algorithm adds the resulting state to the bank of good states. When all transitions are executed from all N-step states, the algorithm logs the number of good states and starts analyzing the N+1-step states.

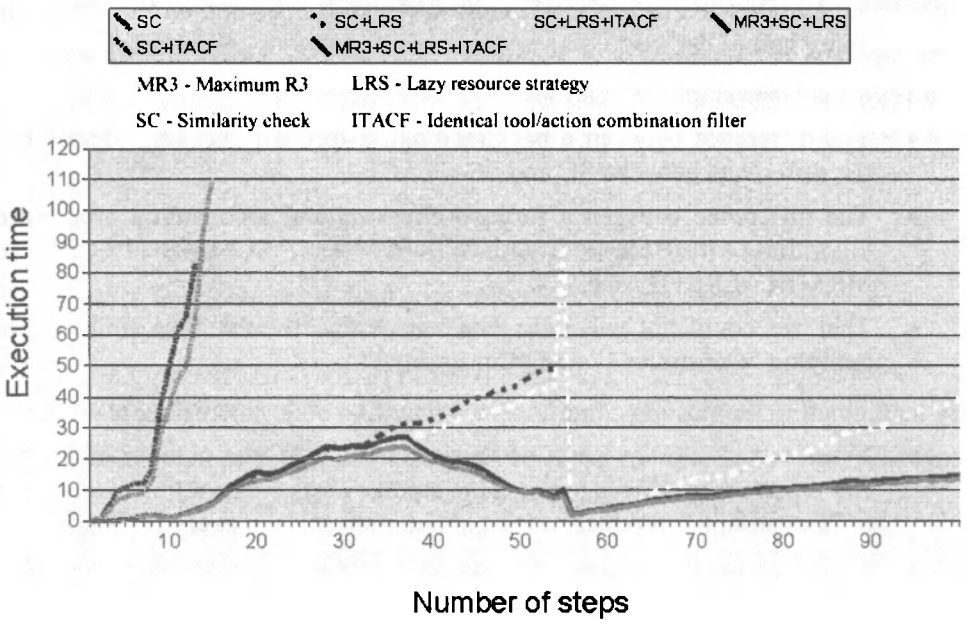


Figure 5: Impact of constraints on execution time

The algorithm was executed on a middle-class multi-user server several times with different constrains enabled and with the maximum execution time of 120 seconds/step. The results are summarized in Figure 5 (some results were dropped to increase readability).

As we expected, the algorithm could not solve the problem without constraints in the available time. At first glance, it may be a surprise to find that the algorithm runs out of memory or execution time in two to five steps only, if we do not apply any non-internal constraints. On second thought, four steps mean about 16 million valid states from the 69 billion possible states. Even with the SC and ITACF constraints enabled, execution time still increases like an exponential curve, but we may reach even step 15. If we add LRS (and thus the maximum R1 and R2 constraints named "Constraint 1" and "Constraint 2" in the paragraph "Constraints"), the problem becomes solvable in the limited time. There is, however, a spike in the execution time at the 56th TU, which is the result of the enormous number of rollbacks caused by not having enough R3s in the store. It even halts SC+LRS, which would need about 140 seconds of execution time for this step. The MR3 constraint (which incorporates "Constraint 3" and "Constraint 4" from the paragraph "Constraints") filters out this spike, reduces execution time

from about 40 steps and halves total execution time. It does not, on the other hand, help anything before the 31st TU.

In this particular example, we can see

- The difference between a backtrace and a forward checking algorithm in the difference between SC and SC+ITACF.
- The difference between a forward checking and a constraint propagation algorithm in the difference between SC+LRS+ITACF and MR3+SC+LRS+ITACF.
- That we could not solve the problem within the set time limit without applying "Constraint 1" and "Constraint 2"

By manipulating the starting conditions, we may find other interesting results:

- If we set the storage costs of the resources to 0, the number of possible states increases. This is because the storage costs create a hidden constraint of "Constraint 5: produce at least about 1000 MUs in every 20 TUs"
- By increasing the starting amount of MUs, "Constraint 5" can be weakened or eliminated.
- By decreasing the starting amount of R1, "Constraint 5" and LRS can be weakened.
- Strong constraints (the ones that reduce the number of possible states considerably) often disqualify the same states and thus they are rarely additive.

9. CONCLUSION

Even with the ever-increasing computing power available today, there are problems that still cannot be solved in a human lifetime. These problems have very high algorithmic complexity. Industrial optimization problems (especially problems from the domain of logistics, manufacturing and scheduling) are often such problems. Constraint programming is an effective tool to reduce the complexity of such problems. In this article, a problem is explained and its complexity is analyzed. After adding constraints to the problem, the formerly unsolvable problem is solved by a tool which is developed by the author and is used in the education, thus the complexity reduction effect of constraints is demonstrated. Equalities of constraints and non-constraint methods – in the terms of complexity reduction – are also demonstrated.

REFERENCES

[1] RSA SECURITY INC, *Has the RSA algorithm been compromised as a result of Bernstein's Paper?*, http://www.rsasecurity.com/rsalabs/node.asp?id=2007

[2] CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L.: *Introduction to Algorithms*, MIT Press, 1990

[3] PER J. KRAULIS: *Algorithmic complexity*, http://www.sbc.su.se/~per/molbioinfo2001/multali-algocomplex.html

[4] MICHAEL L.: *Algorithms & Data Structures , Sorting Algorithms*, http://linux.wku.edu/~lamonml/algor/sort/sort.html

[5] HOMER COMPUTER SERVICES PTY LTD, *Glossary of terms*, http://www.homercomputer.com.au/homer_software_guide/glossary.htm

[6] BLACK, P.E.: *Breadth-first search*, http://www.nist.gov/dads/HTML/breadthfirst.html

[7] KRZYSZTOF A.: *Principles of Constraint Programming*, Cambridge University Press, 2003