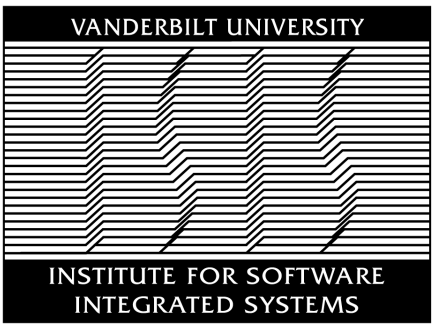


Institute

for

Software-Integrated Systems

Technical Report

TR #: **ISIS-01-205**

Title: **Analysis and Representation of Clauses in Satisfiability of Constraints**

Authors: **Jonathan Sprinkle, Chris van Buskirk, Gabor Karsai**

DARPA Grant No. F30602-99-2-050

Introduction

The scheduling of tasks and resources across a time interval is a non-trivial problem. Firstly, there are constraints on the tasks (a certain duration, must run after another task, etc.), and then the resources that are used during the execution of the task must be available at the correct times. Add to this that the tasks may occur only during specified intervals (time windows) within the planning horizon, and the comprehension of the complexity of the problem has begun.

One approach to solving this problem is to encode the constraints of the tasks and resources in accordance with the format of some paradigm independent constraint solver. One such category of constraint solver is a satisfiability program that expects the encoding of the constraints in conjunctive-normal format (CNF). CNF is a boolean product of sums, and the solution of the problem is the solution to each individual sum in the entire encoding. Thus, either a solution is present, or not.

The crux of this problem is not finding a way to encode the constraints – that problem is readily solved. The problem is in encoding the constraints such that the complexity of the encoding scales well. "Well" in this case means that the complexity does not increase exponentially in time (time of the encoding of the problem, or determination of a solution) or size (memory needed for the solution).

The purpose of this paper is to examine the methods used to find a solution to this problem, and explain the rationality behind them. This paper explores the definition of scheduling constraints, how they are realized in the constraint solver, and the complexity/size analysis of the scaling of the constraints.

Brief introduction to the constraint solver

As previously mentioned, the constraint solver takes a product of sums (POS) expression. An example of a POS expression is,

$$(A \vee B \vee C) \wedge (\neg C \vee D \vee A) \wedge (\neg D \vee E)$$

The constraint solver considers this as three *clauses*. A clause is a boolean sum of *variables* (e.g. A, B, C) which may be written as true or not (not is denoted by the "¬" character in the above expression). Note that the POS expression excludes the use of parentheses to cleverly write an expression, so use of DeMorgan's rule for rewriting the expression in fewer clauses gives no advantage.

After it looks for a solution, the constraint solver returns an indicator as to whether or not a solution was found. If a solution *was* found, then the output of the constraint solver is the set of variables, each of which is assigned a true or false value. For example, in the above statement, the output of the solver might be something like this,

A 1

B 0

C 0

D 0

E 1

Examination of the three clauses reveals that (when 1 is taken to be "true") the expression evaluates as true. Encoding the clauses is the problem examined in this paper. As a general rule, the more clauses there are to an encoding, the longer it takes to find a satisfaction of that encoding. Therefore, it is to the user's advantage to find a terse encoding, and to develop an algorithm to efficiently create the encoding.

Framing the problem

The specific problem that demands a solution is the scheduling of resources (tools and personnel) to perform tasks.

Tasks

The tasks must be performed in a window of time (i.e. a task must start after time j_0 and end before j_1) and they may be constrained to follow or precede other task(s) (i.e. T_1 precedes T_0). Tasks are also defined to require a certain set of resources. For each task, there is a definite duration and (when scheduled) a start and end time.

Resources

A resource is a generic name for a physical entity required to complete a task. The two types of physical entities required for this scheduling problem are tools and personnel. However, these are abstracted as simple resources for purposes of the encoding of the problem.

Variables

In order to encode the constraint relationships between resources and tasks, several types of variables are used. In keeping with the constraints of the tasks, there are defined the Ends Before (EB), Starts After (SA), and PRecedes (PR) variables.

The other important class of variables that is used is the resource consumption variable, or Running (R) variable. For each resource required by a task, and for each time it can be in use, there is an R variable. This will be discussed in more detail.

The final variable used in conjunction with this paper is the Limit (L) variable. The L variable is the subject of most of the complexity analyzed in this paper.

Time

The notion of time in a boolean constraint solver is, not surprisingly, discrete and not continuous. Before encoding the problem, an agreed "timeslot" value is construed which represents an interval of time, e.g. 60 minutes. An example of the discrete timeslot is that timeslot 6 is the interval of time that is [300-360) minutes after the beginning time.

The coherence of the encoding

Each type of variable has its own semantics (and for the purpose of concise explanation, a syntax also). Refer to Table 1 for this information. Moreover, since the boolean solver has no semantic knowledge of the difference between an EB variable and a PR variable, these semantics must be present in the encoding itself; this is the enforcement of the coherence of the constraint. An explanation of the coherence needs and implementations follows for each variable type.

Table 1 - Syntax and Semantics of Variables

	Syntax	Semantics
Starts After	**SA(T,j)**	**Task T starts after time j**
Ends Before	**EB(T,j)**	**Task T ends before time j**
Precedes	**PR(T_0,T_1)**	**Task T_0 must complete before Task T_1 may begin**
Running	**R(T,j,k)**	**Task T is running at time j while using resource k**
Limit	**L(j,x,y)**	**At time j, if you examine any x Tasks, then y or fewer of them are**

		actually running

In general, *j* stands for a timeslot, and *T* represents a particular Task (sometimes denoted by T_0, T_1, etc.).

The SA, EB, and PR variables are taken from the work of Crawford and Baker '94 [1]. These variables were used after exploring the work of [1], and then the variable set extended when the management of resources became important. More observations on the usage of [1] appear in the **Conclusions and future work** section of this paper.

Starts After

There are four coherence requirements for the SA variable.

(1) $SA(T,j_0+1) \rightarrow SA(T,j_0)$ — If a task T starts after j_0+1, then it must also start after j_0

(2) $SA(T,j_0) \rightarrow \neg EB(T,j_0+dur_T-1)$ — If a task T starts after j_0, then it cannot end until the duration of the task has expired

(3) $SA(T_0,j_0) \wedge PR(T_0,T_1) \rightarrow SA(T_1,j_0+dur_T)$ — If a task T_0 starts after j_0, and task T_0 precedes task T_1, then task T_1 must start after task T_0's duration has expired

(4) $SA(T,j_0) \rightarrow \neg R(T,0 \ldots j_0)$ — A task T that starts after j_0 may not run at any time before j_0

Ends Before

There are two coherence requirements for the EB variable.

(5) $EB(T,j_0) \rightarrow EB(T,j_0+1)$ — If a task T ends before j_0, then it must also end before j_0+1

(6) $EB(T,j_0) \rightarrow \neg R(T,j_0 \ldots j_8)$ — A task T that must end before j_0 may not be running at any time j_0 or afterward

Precedes

There is only one coherence constraint for the PR variable. Note that this is also listed as #3 in the SA variable list.

(7) $SA(T_0,j_0) \wedge PR(T_0,T_1) \rightarrow SA(T_1,j_0+dur_T)$ — If a task T_0 starts after j_0, and task T_0 precedes task T_1, then task T_1 must start after task T_0's duration has expired

Running

(8) $R(T,j_0) \wedge \neg R(T,j_0-1) \rightarrow R(T,j_0+1..j_0+dur_T-1)$ — Time j_0 is recognized as the start of the task T, and enforces the constraint for all time slots of the task to be running

(9) $R(T,j_0) \wedge \neg R(T,j_0-1) \rightarrow \neg R(T,j_0+dur_T)$ — Time j_0 is recognized as the start of the task T, and enforces that the task end at the proper time

(10) $R(T,j_0) \wedge \neg R(T,j_0+1) \rightarrow \neg R(T,j_0+2 \ldots j_8)$ — Time j_0 is recognized as the end of the task T, and the task will not start again

(11) $R(T,j_0) \vee R(T,j_0+1) \ldots \vee R(T,j_{N-1})$ — Ensures that the task T must run at some point in the N time slots of the entire time table

The limit

The limit variable (L) is the most complex variable in the entire encoder. Its responsibility is to manage the constraint that: tasks are only scheduled so that they have resources available for them.

Thus, a new sort of coherence must be established – only as many tasks (requiring a resource *y*) as there are *y*'s may be scheduled at the same time. This is accomplished through the L variable, as evidenced in the following example.

$$L(j,x,y) \rightarrow L(j,x\text{-}1,y\text{-}1) \vee [L(j,x\text{-}1,y) \wedge \neg R(x,j)]$$

Where *x* is the number of tasks that require *y*'s at time *j*, and y is the number of *y*'s that exist. What does this mean? Well, stated most simply, L(*j,x,y*) should be read, "At time *j*, counting *x* tasks results in *y* or fewer that are actually running."

The implication of this statement is that there are two ways to achieve this task. One is that if you count *x*-1 tasks, then *y*-1 or fewer are running. The second way is that in counting *x*-1 tasks that *y* or fewer are running, *but* that the x^{th} task is most assuredly *not* running.

There can be some simplification done here,

$$L(j,x,y) \rightarrow L(j,x\text{-}1,y\text{-}1) \vee [L(j,x\text{-}1,y) \wedge \neg R(x,j)]$$

$$L(j,x,y) \rightarrow [L(j,x\text{-}1,y\text{-}1) \vee L(j,x\text{-}1,y)] \wedge [L(j,x\text{-}1,y\text{-}1) \vee \neg R(x,j)]$$

Note that,

$$L(j,x\text{-}1,y\text{-}1) \vee L(j,x\text{-}1,y) \leftrightarrow L(j,x\text{-}1,y)$$

from the definition of L, since it is easier to have *y* or fewer tasks running than it is to have *y*-1 or fewer tasks running. Hence,

$$L(j,x,y) \rightarrow L(j,x\text{-}1,y) \wedge [L(j,x\text{-}1,y\text{-}1) \vee \neg R(x,j)]$$

So, what is the coherence implicit here (if any), and how to encode it? Also, how many clauses will this result in? These are the questions addressed in the remainder of the paper.

The cost of capture

Firstly, it is important to examine the discontinuities in the definition of the L variable. This is because the definition at these discontinuities defines the L variable at all other points in its domain. Secondly, how large is the space of the clauses for some of the cases of x and y, and is there a closed form formula for the calculation of the number of these clauses? Lastly, what is the most efficient way to generate the clauses during the encoding of the problem?

Recall that the constraint solver requires the clause to be formatted in CNF. The conversion of A → B into CNF format is given by,

$$A \rightarrow B = B \vee \neg A$$

Conversions are given for each individual constraint, and a final representation of how the clause is actually written (unless specified otherwise, the relationship is 1:1, and no conversion is necessary). All calculations are based upon the A → B representation, and then multiplied by the number of CNF clauses necessary to represent the A → B clause. Each coherence constraint and its appropriate formula are listed below.

Starts After and Precedes

$SA(T,j_0+1) \rightarrow SA(T,j_0)$

Each instance of the constraint results in exactly one clause. Summing the number of timeslots and again the number of tasks will determine the number of clauses generated. Therefore,

$$\sum_{t=1}^{T}\sum_{j=2}^{J}1$$

where T is the number of tasks (each of which has exactly one Starts After constraint), and there are J total timeslots.

$SA(T,j_0+1) \rightarrow \varnothing EB(T,j_0+dur_T-1)$

Each instance of the constraint results in exactly one clause. Summing the number of timeslots and again the number of tasks will determine the number of clauses generated. Therefore,

$$\sum_{t=1}^{T}\sum_{j=1}^{J-dur_t}1$$

where T is the number of tasks (each of which has exactly one Starts After constraint), dur_t is the duration of task t, and there are J total timeslots.

$SA(T_0,j_0) \grave{U} PR(T_0,T_1) \rightarrow SA(T_1,j_0+dur_{T0})$

Each instance of the constraint results in exactly one clause. Therefore, summing the number of instances will determine the number of clauses generated. Thus,

$$\sum_{t=1}^{T}\sum_{j=1}^{J-dur_t}PR_t$$

where T is the number of tasks, PR_t is the number of Precedes constraints for each task t, J is the total number of timeslots, and dur_t is the average duration of the tasks.

$SA(T,j_0) \rightarrow \varnothing R(T,0\ldots j_0)$

Each instance of the constraint results in exactly one clause. Therefore, summing the number of instances will determine the number of clauses generated. Thus,

$$\sum_{t=1}^{T}\sum_{j=2}^{J}\sum_{k=1}^{j-1}1$$

where T is the number of tasks, and J is the number of timeslots.

Ends Before

$EB(T,j_0) \rightarrow EB(T,j_0+1)$

Each instance of the constraint results in exactly one clause. Therefore, summing the number of instances will determine the number of clauses generated. Thus,

$$\sum_{t=1}^{T}\sum_{j=1}^{J-1}\sum_{k=j+1}^{J}1$$

where T is the number of tasks, and J is the number of timeslots.

$EB(T,j_0) \rightarrow \varnothing R(T,j_{0..}j_8)$

Each instance of the constraint results in exactly one clause. Therefore, summing the number of instances will determine the number of clauses generated. Thus,

$$\sum_{t=1}^{T}\sum_{j=1}^{J}\sum_{k=j}^{J}1$$

where T is the number of tasks, and J is the number of timeslots.

Running

For the Running variable, the calculations are easier, but result in the most clauses. This is because to guarantee that once a task starts that it runs for its duration requires a constraint for each timeslot. One major difference between the R and SA/EB variables is that the R variable is defined only on the domain of timeslots that the task could actually be running, whereas the SA/EB variables are defined for the entire planning window. Thus, all of the time domain calculations for the R variable are from j_0 (the earliest possible run-time) to j_X (the latest possible run-time).

Each running variable calculation is for one resource only. Therefore, for the complete calculation, it is necessary to multiply each total sum of running variables by the number of resources that are subject to that constraint.

$R(T,j_0)$ Ù Ø$R(T, j_0 - 1) \rightarrow R(T,j_0+1 \ldots j_0+dur_T-1)$

Here, j_0 has been recognized as the beginning of the task (since the task is not running in the previous timeslot), and this implies that the task should be running until its duration has expired. The number of clauses necessary to completely express this is reflected through the following induction.

Let j_0 be the first timeslot in which T could start, and j_X be the last timeslot by which T must end
Now, the task is definitely not running at time j_0-1, since that time is not valid for the task.
Therefore there will be dur_T-1 different values for the implication of the clause.
Now, let $j_1=j_0+1$, then there are dur_T-1 different implication values here, also.
…
Now, let $j_n=j_0+n$, then there are dur_T-1 different implication values here.
…
Now, let $j_y=j_0+y$, and $j_y=j_X-dur_T$, then this is the last possible starting time.

Therefore, the total number of necessary clauses is,

$$\sum_{t=1}^{T}\sum_{j=j_0}^{j_X-dur_t}(dur_t-1)$$

$R(T,j_0)$ Ù Ø$R(T, j_0 - 1) \rightarrow$ Ø$R(T, j_0+dur_T)$

Here, j_0 has been recognized as the beginning of the task (since the task is not running in the previous timeslot), and therefore the task should be completed by time j_0+dur_T. The number of clauses necessary to completely express this is demonstrated in the following induction.

Let j_0 be the first timeslot in which T could start, and j_X be the last timeslot by which T must end
Now, the task is definitely not running at time j_0-1, since that time is not valid for the task.
Therefore there will be *1* value for the implication of the clause.
Now, let $j_1=j_0+1$, then there is *1* implication here, also.
…
Now, let $j_n=j_0+n$, then there are *1* different implication values here.
…

Now, let $j_y=j_0+y$, and $j_y=j_X-dur_T$, then this is the last possible starting time.

Therefore, the total number of necessary clauses is,

$$\sum_{t=1}^{T} \sum_{j=j_0}^{j_X-dur_t} 1$$

$R(T,j_0) \wedge \varnothing R(T,j_{0+1}) \rightarrow \varnothing R(T,j_0+2 \ldots j_8)$

Here, j_0 has been recognized as the ending of the task (since the task is not running in the following timeslot), and therefore the task should not run again. The number of clauses necessary to completely express this is.

Let j_0 be the first timeslot in which T could start, and j_X be the last timeslot by which T must end
Since T can not end until at least $j = j_0+dur_T$, then no clauses need be created until that time
Therefore there will be *1* value for the implication of each clause
Now, there will be one clause for each timeslot from j_0+dur_T until j_X.

Therefore, the total number of necessary clauses is,

$$\sum_{t=1}^{T} \sum_{j=j_0+dur_t-1}^{j_X} \sum_{k=j+2}^{j_X} 1$$

$R(T,j_0) \wedge \varnothing R(T, j_0 -1) \rightarrow \varnothing R(T, j_0+dur_T)$

Here, j_0 has been recognized as the beginning of the task (since the task is not running in the previous timeslot), and therefore the task should be completed by time j_0+dur_T. The number of clauses necessary to completely express this is demonstrated in the following induction.

Let j_0 be the first timeslot in which T could start, and j_X be the last timeslot by which T must end
Now, the task is definitely not running at time j_0-1, since that time is not valid for the task.
Therefore there will be *1* value for the implication of the clause.
Now, let $j_1=j_0+1$, then there is *1* implication here, also.
…
Now, let $j_n=j_0+n$, then there are *1* different implication values here.
…
Now, let $j_y=j_0+y$, and $j_y=j_X-dur_T$, then this is the last possible starting time.

Therefore, the total number of necessary clauses is,

$$\sum_{t=1}^{T} \sum_{j=j_0}^{j_X-dur_t} 1$$

The Limit

To calculate the cost of the encoding of the L variable is quite a bit more complicated than any of the others. Fortunately, though, careful calculation and engineering results in a complex encoding that is extremely efficient in space and resource usage. The L's used in this portion of the paper omit the '*j*' (time) parameter for brevity, and refer only to *x* and *y*.

The atomic definition

For values of x and y, the L variable reduces to a simple expression of $\vee$'s and $\wedge$'s. One obvious example of this is the definition of L(0,0), which is always true. L(x_0,x_0) is the more general case of this.

In the case where y is greater than x, then L(x,y) will always be true, since it is impossible for more tasks to be running than x. In the case where y is less than 0, then L(x,y) will always be false, as it is not possible to have a negative amount of tasks running.

Consider then, the following definition of L(x,y):

$$L(x, y) = \begin{cases} 1 & x <= y \\ 0 & y < 0 \\ L(x-1, y) \wedge [L(x-1, y-1) \vee \neg R(x)] & \text{else} \end{cases}$$

Now, it is possible to build up a library of clauses, from these atomic definitions.

$$\begin{aligned} L(1,0) &= L(0,0) \wedge (L(0,-1) \vee \neg R(1)) \\ &= 1 \wedge (0 \vee \neg R(1)) \\ &= \neg R(1) \end{aligned}$$

This makes sense, as the only way for L(1,0) to be satisfied is if task 1 is not running. It turns out that any definition of L(x,0) turns out to be the x R's anded together, as shown,

$$L(x,0) = \neg R(1) \wedge \neg R(2) \wedge \ldots \wedge \neg R(x)$$

Some other examples of the L definition in terms of the R variables is as follows:

$$\begin{aligned} L(1,1) &= 1 \\ L(2,0) &= L(1,0) \wedge (L(1,-1) \vee \neg R(2)) \\ &= \neg R(1) \wedge \neg R(2) \\ L(2,1) &= L(1,1) \wedge (L(2,0) \vee \neg R(2)) \\ &= 1 \wedge ((\neg R(1) \wedge \neg R(2)) \vee \neg R(2)) \\ &= (\neg R(1) \vee \neg R(2)) \wedge (\neg R(2) \vee \neg R(2)) \\ &= \neg R(1) \vee \neg R(2) \\ L(2,2) &= 1 \\ L(3,0) &= \neg R(1) \wedge \neg R(2) \wedge \neg R(3) \\ L(3,1) &= L(2,1) \wedge (L(2,0) \vee \neg R(3)) \\ &= (\neg R(1) \vee \neg R(2)) \wedge ((\neg R(1) \wedge \neg R(2)) \vee \neg R(3)) \\ &= (\neg R(1) \vee \neg R(2)) \wedge (\neg R(1) \vee \neg R(3)) \wedge (\neg R(2) \vee \neg R(3)) \\ L(3,2) &= L(2,2) \wedge (L(2,1) \vee \neg R(3)) \\ &= L(2,1) \vee \neg R(3) \\ &= \neg R(1) \vee \neg R(2) \vee \neg R(3) \\ L(3,3) &= 1 \end{aligned}$$

Calculation of the number of clauses for L(x,y)

Without expanding each instance of L(x,y), we can actually determine a closed form solution for the number of clauses that each will use. Note that L(2,0) has 2 clauses (two variables anded together), L(2,1) has only one clause, L(3,0) has three clauses, L(3,1) has 3, and L(3,2) has 1.

Placing the L's as follows,

L(0,0)

L(1,0) L(1,1)

L(2,0) L(2,1) L(2,2)

L(3,0) L(3,1) L(3,2) L(3,3)

L(4,0) L(4,1) L(4,2) L(4,3) L(4,4)

Now, place the number of clauses each generates in the same position,

1

1 1

2 1 1

3 3 1 1

???? ? ?? ??? ????

This bears a remarkable semblance to the binomial tree,

1

1 1

1 2 1

1 3 3 1

However, it is offset by one digit. In fact, the proof of this is quite simple. Take L(x,y). Now,

$$L(x,y) = L(x\text{-}1,y) \wedge (L(x\text{-}1,y\text{-}1) \vee \neg R(x))$$

So, the number of clauses in L(x,y) will be the number of clauses in L(x-1,y) plus the number of clauses in L(x-1,y-1) (since the sum (L(x-1,y-1) $\vee$ $\neg$R(x)) results in $\neg$R(x) being "added" to each member of each "and" of the L(x-1,y-1) statement). Therefore,

$$\sum L(x,y) = \sum L(x\text{-}1,y) + \sum L(x\text{-}1,y\text{-}1) + 1$$

Since the $\wedge$ between the two L's creates another clause. Recall that the definition of a node in the binomial tree is

$$N(x,y) = N(x\text{-}1,y) + N(x\text{-}1,y\text{-}1)$$

The formula for value of the n^{th} place in a binomial tree is given by the choice function, C(x,y), or

$$\binom{x}{y} = \frac{x!}{y!(x-y)!}$$

The formula for the x,y value of the $\sum L(x,y)$ is given as follows,

$$\sum L(x,y) = C(x,y+1)$$

Note that L(x,x) = 1, even though C(x,x+1) is undefined.

Analysis

Having the number of clauses dependent upon the choice function is not a good thing. Consider the case L(50,34) which is C(50,35) = 2.250829E12 clauses. In the interest of decreasing the amount of resources necessary to represent the clauses, different methods were explored.

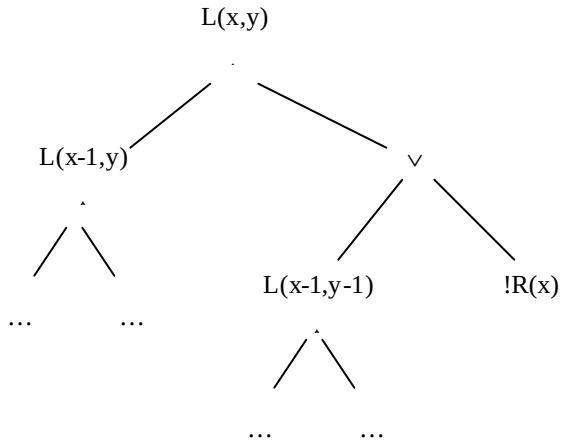
Concise representation of the Limit

Initially, the L variables were semantically enforced through the recursive definition, constantly calling the constructor of the L variable until the atomic definitions were reached. However, this has proved time intensive, and also memory intensive, as so many instances of the variables were created needlessly over and over again.

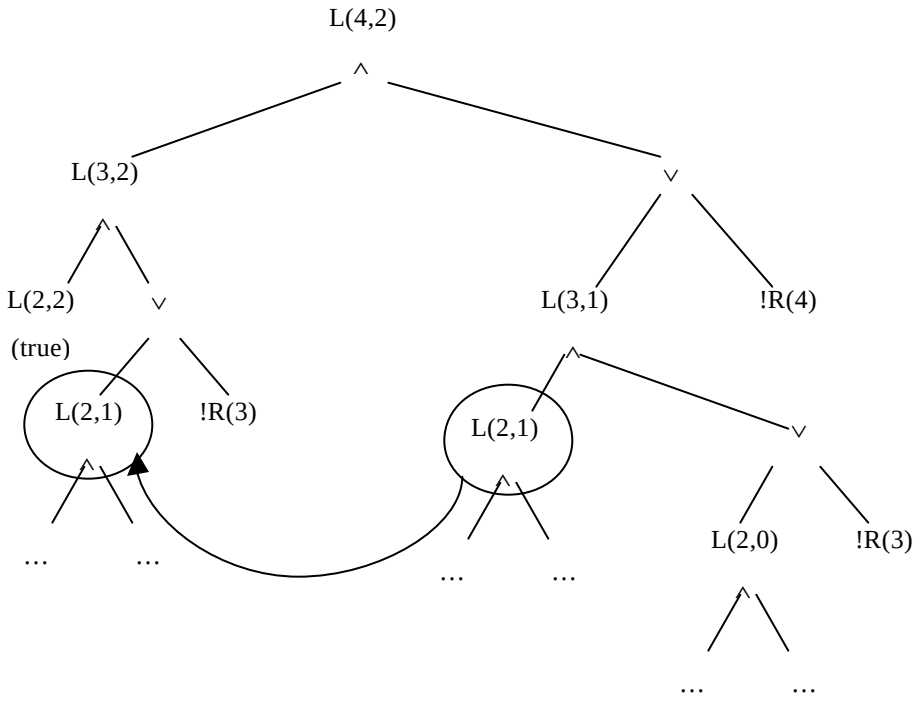
Take for example, the creation of L(3,3,2) [timeslot 3, count 3 and 2 or fewer are running], which recursively defines itself as L(3,2,1) and L(3,2,2), etc. If the same conditions hold for timeslot 4,5,…t_n, then all of those variables would need to be created again. Consider also the L(3,4,2) creation, which eventually would use L(3,3,2) as part of its definition. In all of these cases, a blind reproduction of the variables results in a needless use of CPU cycles and memory.

Consider the more abstract approach of defining the L(*j,x,y*) variable in terms of itself, but holding a static record of the structure in memory, building upon it as necessary, and using the already existing structure to write out the clauses. What benefits would this yield, both in memory usage, and time usage?

The following diagram shows the definitive case of the graph that would be produced (once again, the time parameter is omitted for brevity).



It is also possible that certain L variables will be duplicates within the tree. Consider the case of L(4,2),



In this case, L(2,1) is represented twice in the same graph. Using a pointer type structure, it is possible to reference the location of the L(2,1) node in the graph, and then no further definition is necessary from that node, as it would be redundant information.

Using this tree structure, it is possible to represent all of the possible clauses (a function of the factorial) of an L variable with a memory size that increases only polynomially. The equation for the number of L nodes in the graph is,

$$\sum L(x,y)_{nodes} = \sum L(x\text{-}1,y\text{-}1)_{nodes} + \sum L(x\text{-}1,y)_{nodes}$$

The proof of this is presented in an example. Consider the definition of L(9,6).

$L(9,6) = L(8,5) \vee (L(8,6) \wedge \neg R(9))$

$L(8,6) = L(7,5) \vee (L(7,6) \wedge \neg R(8))$

$L(7,6) = L(6,5) \vee (L(6,6) \wedge \neg R(7))$

$L(6,6) = 1$

$L(8,5) = L(7,4) \vee (L(7,5) \wedge \neg R(8))$

$L(7,5) = L(6,4) \vee (L(6,5) \wedge \neg R(7))$

$L(6,5) = L(5,4) \vee (L(5,5) \wedge \neg R(6))$

$L(5,5) = 1$

$L(7,4) = L(6,3) \vee (L(6,4) \wedge \neg R(7))$

$L(6,4) = L(5,3) \vee (L(5,4) \wedge \neg R(6))$

$L(5,4) = L(4,3) \vee (L(4,4) \wedge \neg R(5))$

$L(4,4) = 1$

$L(6,3) = L(5,2) \vee (L(5,3) \wedge \neg R(6))$

$L(5,3) = L(4,2) \vee (L(4,3) \wedge \neg R(5))$

$L(4,3) = L(3,2) \vee (L(3,3) \wedge \neg R(4))$

$L(3,3) = 1$

$L(5,2) = L(4,1) \vee (L(4,2) \wedge \neg R(5))$

$L(4,2) = L(3,1) \vee (L(3,2) \wedge \neg R(4))$

$L(3,2) = L(2,1) \vee (L(2,2) \wedge \neg R(3))$

$L(2,2) = 1$

$L(4,1) = L(3,0) \vee (L(3,1) \wedge \neg R(4))$

$L(3,1) = L(2,0) \vee (L(2,1) \wedge \neg R(3))$

$L(2,1) = L(1,0) \vee (L(1,1) \wedge \neg R(2))$

$L(1,1) = 1$

$L(3,0) = 0 \vee (L(2,0) \wedge \neg R(3))$

$L(2,0) = 0 \vee (L(1,0) \wedge \neg R(2))$

$L(1,0) = 0 \vee (1 \wedge \neg R(1))$

$L(0,0) = 1$

Now, consider the general case of this definition. For any x, y combination, L(x,y) recursively defines itself until it gets to $L(x_0,y_0)$ where $x_0 = y_0$, or $L(x_0,0)$, where the definition is trivial. In either case, the following numbers are relevant:

$L(x,y) \rightarrow$ produces $x - y + 1$ definitions until $x_0 = y_0$

$L(x_0,0) \rightarrow$ produces x_0 more definitions

So, for any x, y combination, the equation for the number of definitions that exist is,

$$\sum L(x,y)_{nodes} = y(x - y + 1) + x_0$$

since there are y definitions that are $x - y + 1$ in length. Also, note that $x_0 = x - y$, so

$$\begin{aligned} \sum L(x,y)_{nodes} &= y(x - y + 1) + x - y \\ &= yx - y^2 + y + x - y \\ &= x + xy - y^2 \end{aligned}$$

Now, the polynomial size of the number of definitions is apparent.

The final computation necessary is the computation of how many clauses is generated by the definition,

$L(x,y) = L(x\text{-}1,y\text{-}1) \vee (L(x\text{-}1,y) \wedge \neg R(x))$

Note that = is defined as ←→, which means that the implication is in both directions

$\rightarrow$ $L(x,y) \rightarrow L(x\text{-}1,y\text{-}1) \vee (L(x\text{-}1,y) \wedge \neg R(x))$

$L(x,y) \rightarrow (L(x\text{-}1,y\text{-}1) \vee L(x\text{-}1,y)) \wedge (L(x\text{-}1,y\text{-}1) \vee \neg R(x))$
$[(L(x\text{-}1,y\text{-}1) \vee L(x\text{-}1,y)) \wedge (L(x\text{-}1,y\text{-}1) \vee \neg R(x))] \vee \neg L(x,y)$
$[(L(x\text{-}1,y\text{-}1) \vee L(x\text{-}1,y) \wedge \neg L(x,y)] \wedge [L(x\text{-}1,y\text{-}1) \vee \neg R(x) \vee \neg L(x,y)]$
Thus, 2 clauses are required for this direction.

$\leftarrow$ $L(x\text{-}1,y\text{-}1) \vee (L(x\text{-}1,y) \wedge \neg R(x)) \rightarrow L(x,y)$
$L(x,y) \vee \neg [L(x\text{-}1,y\text{-}1) \vee (L(x\text{-}1,y) \wedge \neg R(x))]$
$L(x,y) \vee [\neg L(x\text{-}1,y\text{-}1) \wedge \neg (L(x\text{-}1,y) \wedge \neg R(x))]$
$L(x,y) \vee [\neg L(x\text{-}1,y\text{-}1) \wedge (\neg L(x\text{-}1,y) \vee R(x))]$
$[L(x,y) \vee \neg L(x\text{-}1,y\text{-}1)] \wedge [L(x,y) \vee \neg L(x\text{-}1,y) \vee R(x)]$
Thus, 2 clauses are required for this direction.

For the example case of L(50,35), this encoding results in 650*4 = 2600 clauses, or about 1.15_E-7% of the size of the encoding described by the choice calculation.

Deviation from Crawford and Baker '94

While exploring the scalability and speed of solutions of the encoding, several modifications to [1] were made. In [1], the solution is determined through the PR variable, and a set of true PR's set forth a schedule window that was guaranteed to work. This worked well in the case of many timeslots but few resources.

However the current problem required solutions where many resources were required by many tasks. To do this required the creation of a PR variable for each combination of task and timeslot, resulting in *J*(T-1)*T* variables of type PR. The presence of all of these PR constraints increased the number of clauses generated as in implication (3). By removing the PR constraints the size of one example encoding was about 20% of the original size, and had the same results.

Conclusions and future work

After exploring the solutions to the resource allocation problems in this manner, conclusions are apparent in scalability of the problem, and the advantages of the encoding.

Scalability

Sample data verify the polynomial nature of the encoding. This is presented in Figures 1 and 2. For small problems, the large amount of memory used (shown in Figure 2) is present because the small number of tasks resulted in arrival at a solution well before the garbage collector could run. Therefore, the increased memory is the previous solutions that are awaiting disposal.

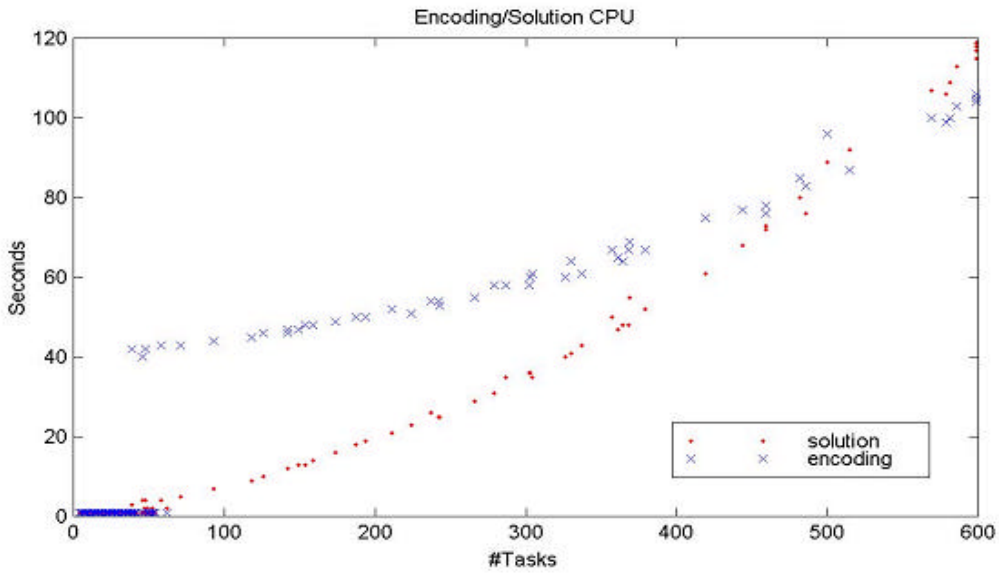


Figure 1

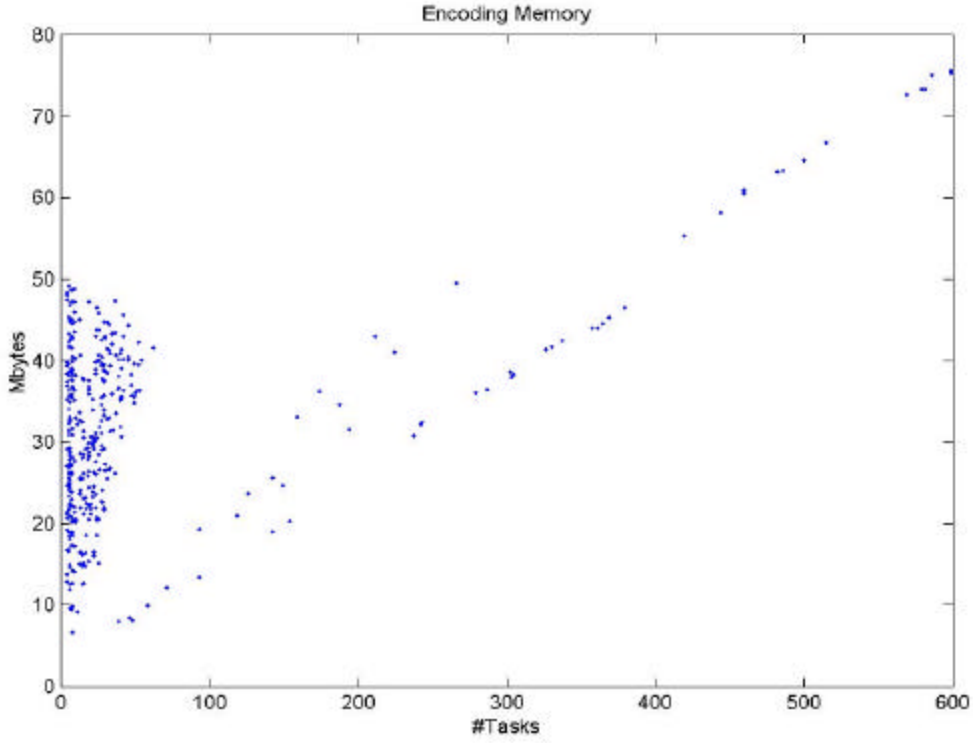


Figure 2

Advantages of this encoding

Obviously, the major advantage of the encoding discussed in this paper is the scalability of the problem. That is, when increasing the number of tasks, the memory required to solve the problem increases linearly, while the encoding increases polynomially.

Also, the memory and encoding sizes increase linearly when increasing the planning horizon (number of timeslots). This is due to the L variable, and the fact that each timeslot contains the same number of L variables, so that as the problem increases in time, that a linear amount of clauses are added.

Future work

After extending the work described in [1], it became apparent that all of the variables described in the Crawford and Baker paper were not required to solve the class of problems under study for the associated DARPA research project. In fact, it is possible that the SA, EB, and PR variables might be used only as a guideline for the creation of R and L variables, and that these latter variables are the only ones required by the solver to find a solution. However, there still remains much exploration and experimentation to prove this.

References

[1] Crawford, J. and Baker A. "Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems." 1994.