

600.325/425 — Declarative Methods
Assignment 2: Constraint Programming
for Planning and Scheduling*

Spring 2012
Prof. J. Eisner

Due date: Friday, March 16, 2 PM

In this assignment, you will gain familiarity with encoding real-world problems as instances of constraint programming, and get a feel for how constraint programming systems work in practice.

Academic integrity and collaboration: You should work on your own for this assignment. As always, the work you hand in should be your own, in accordance with the regulations at <http://cs.jhu.edu/integrity-code>.

How to hand in your work: Specific instructions will be announced before the due date. Your programs have to run in ECLⁱPS^e, which is installed on the `ugrad` machines. Any of those machines that you have access to you may develop on.

Besides the comments you embed in your source code, include all other notes, as well as the answers to the questions in the assignment, into a file named `README`. Include directions for building the executables, either in your `README` or in a `Makefile`.

325 vs. 425: Problems marked “**425**” are only required for students taking the 400-level version of the course. Students in 325 are encouraged to try and solve them as well, and will get extra credit for doing so.

Data: All the files you will need for this project are available in `/usr/local/data/cs325/hw2/` on the `ugrad` machines.

Running ECLⁱPS^e: ECLⁱPS^e is available as `eclps` on all the `ugrad` machines. (Also on the graduate Solaris machines. It could be added to the graduate Linux machines if needed.)

There is a good deal of documentation available at <http://www.eclipseclp.org>. For example, documentation for all of ECLⁱPS^e's built-in predicates is available at <http://www.eclipseclp.org/doc/bips/kernel>, and documentation of constraint libraries such as `ic` and `edge_finder` is at <http://www.eclipseclp.org/doc/bips/lib/>. You may find this helpful in writing your programs.

In this assignment, you will learn to use ECLⁱPS^e, a powerful constraint programming language. ECLⁱPS^e is built on top of the logic programming language Prolog, and as such has all the power of

*Thanks to John Blatz for co-authoring this assignment.

a full-fledged programming language—functions, loops, recursion, data structures, and everything. However, since we don't want to force you learn all the details of logic programming (yet) just to do this assignment, we're only going to ask you to work using a very simple subset of ECLⁱPS^e commands.

Remember in class that we listed three ways to add power to a little language: by expanding its syntax, by embedding it in a more powerful language, or by using another language to write programs in it. We'll be taking the third approach on this assignment, although that is not a requirement. If you know Prolog or are anxious to learn it, then by all means avail yourself of the documentation on the course website (listed under "Resources" for the constraint programming unit), and write whatever code you can get to run.

1. Industrial planning and scheduling is an important real-world application of constraint programming. This broad class of problems seeks to find the optimal ordering of tasks, subject to a variety of constraints on the ordering and conditions necessary to complete the tasks. You can read about the different varieties of problem at <http://www.sciencedirect.com/science/article/pii/S0377221798002045> (free to read from within JHU).

For example, suppose that you are trying to grill bratwurst before settling down to watch the Bears game. You have to complete the following subtasks, with the time, precedence, resource, and labor constraints listed below:

- *defrost sausages*, takes 2 min, requires microwave
- *preheat grill*, takes 20 min, requires grill
- *dice onions*, takes 3 min, requires knife and cutting board
- *toast buns*, takes 1 min, requires grill, grill must be preheated first
- *grill sausages*, takes 10 min, requires grill, grill must be preheated first, sausage must be defrosted first, sausage must be pan-broiled first
- *add sauerkraut, mustard, and onions*, takes 1 min, sausage must be grilled first, sauerkraut must be pan-broiled first, onions must be grilled first, buns must be toasted first
- *grill onions*, takes 8 min, requires grill, grill must be preheated first, onions must be diced first
- *pan-broil sausage and sauerkraut in beer*, takes 15 min, requires stove, sausage must be defrosted first

It's a small grill, so you can only have one thing on it at a time.

We have provided an ECLⁱPS^e program `bratwurst.ecl` which will find the optimal ordering of subtasks and return the minimum amount of time required to complete them all. The syntax of this program is fairly straightforward; take a look at it and make sure that you basic understanding of what it does.

Run this program in ECLⁱPS^e by doing the following:

- (a) Log on to one of the `ugrad` machines and change to directory `/usr/local/data/cs325/hw2/`.
- (b) Run "`ec1ps`" at a command prompt to start up ECLⁱPS^e.
- (c) Execute the command "`compile('bratwurst.ecl')`."

- (d) Find a solution by typing “`schedule(EndTime)`.”. This will print the value of `EndTime` that satisfies all the constraints listed in `schedule`.

Alternatively, you can run the whole thing from the command line with the “batch” command `eclps -b bratwurst.ecl -e 'schedule(EndTime)'`, and you can redirect this output to a file or program of your choice in the usual way. However, ECL^iPS^e will then run rather quietly and you won't see progress messages as it works.

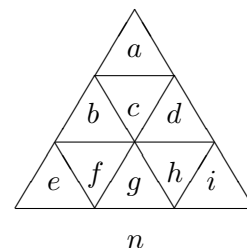
If kickoff is at noon, at what time should you start preparing your bratwurst so that you don't miss any of the game? **Report the answer to this question in your README.**

Note: If you have trouble understanding how this program works, please come see the TA or professor ASAP. You'll need to write your own (very similar) program later in the assignment. The purpose is for you to be able to figure out what constraints you need, not to get bogged down in trying to figure out ECL^iPS^e syntax. We'll be studying logic programming later in the course, so we don't expect you to have mastered it yet. You may want to run a couple other examples off the ECL^iPS^e examples page, <http://www.eclipseclp.org/examples>; there's even a section on planning and scheduling. Feel free to discuss the ECL^iPS^e website's examples on the class mailing list!

2. Now that you've actually used ECL^iPS^e , here are a few warm-up problems to get you started writing your own code. Pick **any two (2)** of these five math problems, and write ECL^iPS^e code to solve them. If you are in **425**, you have to solve **four (4)** of them instead. **Hand in your ECL^iPS^e code, and include the answers in your README .**

- (a) The numbers 1 through 9 can be arranged in the triangles labeled a through i illustrated on the right so that the numbers in each of the 2×2 triangles sum to the same value n ; that is

$$a + b + c + d = b + e + f + g = d + g + h + i = n.$$



For what values of n is there a solution to this puzzle?¹

- (b) Given two integers x and y , let $(x||y)$ denote the concatenation of x by y , which is obtained by appending the digits of y onto the end of x . For example, if $x = 218$ and $y = 392$, then $(x||y) = 218392$. Find 3-digit integers x and y such that $6 \cdot (x||y) = (y||x)$.²

By “3-digit integer,” we mean an integer between 100 and 999; that is, the first digit may not be 0. Please ensure that your encoding respects this constraint.

Note that there are two ways to solve this problem: one uses 6 variables, the other uses 2 variables. Try both!

No, ECL^iPS^e doesn't (yet) support constraints on strings. You should figure out how to solve this string concatenation problem with the kinds of constraints you do know about ...

- (c) Find three isosceles triangles, no two of which are congruent, with integer sides, such that each triangle's area is numerically equal to 6 times its perimeter.³

¹ From USAMTS '04-'05, Round 2, Question 1. www.usamts.org.

² From USAMTS '04-'05, Round 3, Question 1 (a). www.usamts.org.

³ From USAMTS '04-'05, Round 3, Question 2. www.usamts.org.

- (d) The number 12148 has a fun feature: The sum of the first four digits equals the units digit. How many EVEN five-digit numbers have this property?⁴
- (e) Find the smallest square number (perfect square) that uses each digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) once and only once.⁵

Hint: If you write a function `solve(X)` that constrains `X` to be a solution of some problem, you can type `solve(X)` at the ECLⁱPS^e prompt to find a solution, and repeatedly press ‘;’ to find more solutions. You can alternatively type “`findall(X, solve(X), L)`.” to return a list in `L` of all solutions, or “`findall(X, solve(X), L), length(L, N)`.”, to return the number of solutions as `N`.

Hint: If there are a lot of solutions, as in problem 2a, then the list `L` will be too long and ECLⁱPS^e will only print the beginning of it followed by “...”. Here are three possible solutions, any one of which is fine:

- `findall(X,solve(X),List), printf("%Dw",[List]).`
This has an explicit formatted print command as in the `bratwurst.ecl` example. The “%Dw” format says to print `List` in full. You can check the ECLⁱPS^e documentation if you want to know more about the `printf` command.
- `setof(X,solve(X),List).`
`setof` is more or less like `findall`, but the resulting `List` is a set: that is, duplicate values of `X` have been eliminated from it. This may make `List` short enough that ECLⁱPS^e will print it in full without special coaxing.
- `solve(X), write(X), nl, fail.`
This funny idiom means: find a solution `X` (using `solve(X)`), print it (`write(X)`), print a newline (`nl`), and then decide that the solution isn’t good enough and backtrack to find another one (`fail`)!
If you prefer, you could replace the `write(X), nl` part by a `printf` command as above.

If you still get a lot of output (particularly in the scheduling problems below), then you can redirect the output of `eclps` to a file. You can then use any tools you like to analyze the output. (E.g., `perl`, or Unix commands like `sort`, `uniq`, and `wc`.)

3. Now let’s look at a larger, real-world problem. Barry Fox and Mark Ringer of the American Association for AI have put together a series of benchmark problems in planning and scheduling, to test how good solvers are. In this section, we will use ECLⁱPS^e to solve some of these problems.

Their description of the problem is on the `ugrad` machines in the file `rcps.pdf`. Read through it, paying particular attention to the description of the data format and to problems 1-3.

You may want to take a look around their website, which is mirrored at <http://cs.jhu.edu/~jason/325/hw2/benchmrx/rcps.html>.⁶ The data files are already on the `ugrad` machines,

⁴ White House Kids Math Challenges, www.whitehouse.gov/kids/math/. Designed by David Rock of UMass-Dartmouth and Doug Brumbaugh of UCF.

⁵ Ibid.

⁶The original URL was <http://www.neosoft.com/~benchmrx/rcps.html>.

in the directory `/usr/local/data/cs325/hw2/`, so you won't need to download them again. If you find the C++ helper files they provide useful, you are free to make use of them.

The idea behind the problem is as follows. You are the manager of a factory, and in order to manufacture some product, there are 575 different subtasks that you need to accomplish. These subtasks are given names of the form `asm_1.step_575`. Each task requires a certain amount of time to complete, certain types of laborers, can only be done using certain machinery (“in a certain ‘zone’ ”), and can only be done after certain other subtasks have been completed (“precedence constraints”). There are limits on the types of laborers available at different times (“labor constraints”), and on the amount of work that can be done in each zone at any given time (“zone constraints”, a.k.a. “resource constraints”). All of these constraints restrict the possible ordering of the tasks, just as in the `bratwurst` example from section 1.

You don't need to turn anything in for this question; just read `rcps.pdf`.

4. Solve the first problem from the benchmark set using ECLⁱPS^e. The data file you'll use is `rcps.data`, and its format is described in `rcps.pdf`. You should be able to do this by simply modifying the constraints listed in `bratwurst.ecl`, although you'll probably want to write a small script to convert the constraints into ECLⁱPS^e format.

For this problem, you need to consider only the precedence constraints in section 2; you may ignore the labor and zone constraints.

A reference solution is provided in `rcps_s1.data`. **Turn in your ECLⁱPS^e code, named `problem1.ecl`, as well as your list of task start times. This should be in a separate file called `problem1.solution`.** You don't need to use the format of `rcps_s1.data`—see the note on time formats below.. **Include in your README a description of what you did, as well as the total time your ordering requires to complete all the tasks.** If you wrote a script to generate the ECLⁱPS^e code, **turn in that script as well.**

Note: The time format of `rcps_s1.data` can be a little hard to understand—times are encoded in the form ‘11/2+03:25’, which means ‘3 hours and 25 minutes into shift 2 of day 11’. There are 60 minutes in an hour, 7 hours and 30 minutes in a shift, and 2 shifts in a day. This format is needed for scheduling problems that have additional constraints that consider how days are divided into shifts.

However, in *this* assignment, we don't care about the shift schedule. (First, we have no constraints that care whether a job is done during day shift or night shift. Second, we have no constraints that require work to be finished during the shift it began: you may interrupt work at the end of a shift and pick up where you left off at the start of the next shift.) So you probably just want to encode times as an integer representing the number of work minutes since the start time (e.g. 11/2+03:25 becomes simply 9655 [= 25 + 3*60 + 450 + 10*2*450]). **You are free to store and turn in your schedule with times recorded as simply an integer number of minutes.**

5. Write ECLⁱPS^e code to solve the second problem from the benchmark set. For this problem, you must still respect the precedence constraints from the previous problem, but now you must make sure that zone occupancy never exceeds the limits listed in section 4 of `rcps.data`. You can do this using the `cumulative` constraint in the `edge_finder` library. You may still ignore the labor constraints.

This is still essentially the same problem as the bratwurst example; there we had constraints on the ordering of tasks and on zone occupancy (only one thing was allowed on the grill at a time).

Try to run your code. If your code uses the same constraints as the bratwurst example, ECLⁱPS^e will be far too slow to solve it. We'll deal with this in the next section.

You **don't need to turn anything in for this section**, since you didn't write code that produces a solution.

6. Why is your program so slow? Isn't this what ECLⁱPS^e is made for? To understand the problem, we'll need to take a look at what is going on behind the scenes.

The culprit is the line `minimize(labeling(AllVars), EndTime)`. This tells ECLⁱPS^e to find a labeling of `AllVars` that makes `EndTime` as small as possible. It does this using the *branch and bound* algorithm: first, it finds any labeling of `AllVars`, and notes the value v_1 of `EndTime`. It now continues with the backtracking search, but first adds a *new constraint* that `EndTime` $<$ v_1 . If it can find another labeling with the added constraint, with some value `EndTime` = $v_2 < v_1$, then it continues backtracking with an added constraint that `EndTime` $<$ v_2 . It continues in this fashion until it cannot find a labeling. The last labeling found is the optimum solution.

In this way it is guaranteed to find the best solution. Now, we know that the general problem of optimal constraint satisfaction is NP-complete, so that should be a tip-off that this algorithm is not always going to be very fast. We're dealing with a real-world problem here, which means that it's too big to be solved using exponential-time algorithms.

Of course, we can get an approximate algorithm from this iterative procedure; after some timeout, simply stop execution and return the best labeling found so far. How do you get ECLⁱPS^e to do this? The easiest way is this hack: `minimize` prints the costs v_1, v_2, \dots of the intermediate solutions it finds.⁷ So just stop execution at some point, let's say after i intermediate solutions, and then solve the problem again with an additional constraint `EndTime` = v_i , this time asking ECLⁱPS^e to find and print out a labeling rather than to minimize `EndTime`.

Unfortunately, this is not good enough; even the last step `labeling(AllVars)` may be unusably slow. Not only can we not find an optimal solution, we can't find a solution at all!

This is because of the way that `labeling` works. As we saw in the animations of graph coloring in lecture, the order in which we select variables for constraint propagation is very important. Without being given a better plan, `labeling` will select the first variable in the list, assign it the lowest value in its current domain, and propagate as far as it can from that variable assignment. When it finishes propagating, it moves to the next variable in the list, assigns it the lowest value in its domain, and propagates. If propagation ever causes a contradiction, it will backtrack to the most recent assignment, and try assigning the next value.

⁷As long as you use ECLⁱPS^e interactively. You should probably avoid the "batch" option discussed earlier (e.g., `eclps -b bratwurst.ecl -e 'schedule(EndTime)'`), as it would suppress these messages.

When ECLⁱPS^e runs slowly for you, this is usually because it is making poor choices of start times for the first few tasks it decides to assign, and takes a while to discover that there is no way for it to arrange the remaining tasks (given the constraint `EndTime < vi`). We saw the same problem in the graph-coloring animations in class.

If you were trying to schedule these tasks by hand, without using ECLⁱPS^e, you would probably do a pretty good job of ordering variables and values, so this wouldn't be much of a problem. Think about how *you* would do it by hand!

Fortunately, ECLⁱPS^e gives you some control over variable and value ordering. For our purposes, the command you want instead of `labeling` is as follows:

```
search(List, 0, Select, Choice, Method, OptionList),
```

where `List` is the list of variables that you are finding a labeling for. `Select` is the strategy for ordering the variables, chosen from, among others, `input_order`, `first_fail`, `smallest`, `largest`, `occurrence`, and `most_constrained`. `Choice` is the strategy for choosing values to assign to variables—you can have it start with the smallest value, the largest value, the middle value, a random value, or a couple other things. `Method` allows you to bound the backtracking in various ways.

Take a look at the full documentation of this predicate to see the other options:

<http://www.eclipseclp.org/doc/bips/lib/ic/search-6.html>. See if you can find a search option that does something similar to the way you would intuitively approach the problem.

Replace your use of `labeling` from the previous problem with some version of `search`, so that your minimization line will be:

```
minimize(search(AllVars, 0, •, •, •, []), EndTime).
```

Experiment with the parameters of `search` until you find a strategy that will allow you to solve the problem. What is the lowest-total-time schedule that you can find that satisfies all the constraints?

Turn in your ECLⁱPS^e program for this question, which you should call `problem2.ecl`. Using the method described above, hand in a list of start times of the tasks for your best ordering in a separate file called `problem2.solution`.

Include answers to the following in your README:

- Describe briefly the constraints you used.
- What parameters did you use in your call to `search()`?
- What was the best total time you were able to find?
- How long did it take you to find that solution on `ugrad2`? (approximate is okay if it's really long)
- **IMPORTANT:** A clear description of how your search method works, and why you thought it would be effective. (Your grade on this assignment will depend less on the quality of the ordering than it does on your ability to demonstrate an understanding of what the constraint solver is doing.)
- **[425 only]** If you weren't constrained by the choices given to you by `search`, how might you want to choose a variable and value ordering to solve this problem most efficiently?

Hint: You can do pretty well even with *Method=complete*, which gives a standard backtracking search as we've studied in class. It's certainly possible to get your first solution (e.g., with time 42383) within about 5 minutes, and several better solutions within 15 minutes. If you're taking a half-hour without outputting anything, you can do better.

We realize it can be frustrating to sit and watch the computer do nothing for 15–20 minutes. It's not our intention to make this painful for you. Try some things and do your best to make it work, but if you can't find an effective strategy, just write in your README what you tried and why you thought it would work, and you'll get partial credit.

7. **[Extra credit]** You may have noticed in the documentation for `search` that you are in fact allowed to create custom variable selection and value choice methods. See if you can implement a heuristic that allows you to achieve a better labeling. There will be a **prize** for the student who finds the best ordering.

If you do this part, submit your code and optimum ordering as `problem2.ec1` and `problem2.solution`, and describe your method and results in your README just as in the previous problem.

So how can you write your own custom heuristics? The standard heuristics supplied to you by ECLiPSe are actually just the names of ECLiPSe functions. For example, one of those variable selection heuristics is the function `largest/2` (where the `/2` is a conventional notation for “with 2 arguments”). The first argument is a particular variable `X` that `search/6` is considering whether to select. The second argument, `Crit`, is used to return a number that indicates how bad it would be to choose `X`. At each decision step of backtracking search, the function `search/6` will call `largest/2` for every variable that it hasn't yet assigned a value to. It will then look at the value of `Crit` returned by `largest/2`, and select the variable for which the returned `Crit` is smallest.

Let's take a look at the actual code for `largest/2`:

```
largest(X, Crit) :-
    get_bounds(X, Lo, Hi),
    Crit is (-1 * Hi).
```

Note that the function declaration has both the input `X` and output `Crit` listed as arguments of the function name. You can think of this as being similar to pass-by-reference in C++, although that's not exactly what's going on, and you'll learn more later on in the course when you study logic programming.

What does the body of the function do? First it calls `get_bounds/3`, which has one input argument and two output arguments, and which simply returns the current range `Lo..Hi` that ECLiPSe considers plausible for values of variable `X` (i.e., the min and max of `X`'s current domain).

Since the function is designed to select the variable with the largest possible value, it is only concerned with the domain's upper bound `Hi`. Remember that `search/6` will choose the variable for which `Crit` is *smallest*; as we want it to pick the variable with the *largest* possible value in its domain, our criterion for a variable `X` is the *negation* of the largest possible value for `X`. For this, we use the syntax `Crit is Expr`, which simply tells the computer to evaluate the arithmetic expression `Expr`, and assign the resulting value to `Crit`.

Good luck!