600.432/632 — Declarative Methods
Homework 2: Constraint Programming
for Planning and Scheduling*

Prof. Jason Eisner — Spring 2026

Due date: Monday, March 2, 2:00 PM

In this homework, 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.

You will learn to use ECL^{i}PS^e, a powerful constraint programming language. ECL^{i}PS^e is built on top of the logic programming language Prolog, and as such has all the power of 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 homework, we're only going to ask you to work using a very simple subset of ECL^{i}PS^e commands.

Instructions

Academic integrity and collaboration: You should work on your own for this homework. As always, the work you hand in should be your ownh, in accordance with the policy on the syllabus.

Running ECL^{i}PS^e: ECL^{i}PS^e is available as `eclps` on all the `ugrad/gradx` machines. There is a good deal of documentation available at http://www.eclipseclp.org. For example, documentation for all of ECL^{i}PS^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.

Handing in your work: You will submit a written part and a programming part on Gradescope:

- **Homework 2 Written:** submit a PDF file with each problem on a separate page.

- **Homework 2 Code:** submit `minipuzzles.ecl`, `problem1.ecl`, `problem2.ecl`, `problem1.solution`, `problem2.solution`. Also submit a `README.md` or `README.pdf` file that contains your discussion of the code. Please try submitting early: Gradescope will run some autograder tests to ensure your code compiles and runs on Gradescope machines. You can replace your submission up until the deadline. Please contact the TA if your code is not running correctly on Gradescope or if you have any concerns about your choice of language.

You are allowed unlimited submissions, but the problems will not be graded until after the deadline.

Data: All files you will need for this homework are available on the `ugrad` and `gradx` machines at `/usr/local/data/cs432/hw-cp`, or at http://cs.jhu.edu/~jason/432/hw-cp/.

Getting familiar with ECL^{i}PS^e

Remember in class that we listed three ways to add power to a little language: by expanding its syntax, by embedding it in a general-purpose language, or by using a general-purpose language to write programs

---

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

in the little language. We'll be taking the last approach here, as we did in the previous homework.[1]

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 access for JHU folks). 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 ECL^{i}PS^e program `bratwurst.ecl` that 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 ECL^{i}PS^e by doing the following:

   - Make sure you are in the `/usr/local/data/hw-cp` directory.
   - Run `eclps` (or better: `rlwrap eclps`) at a command prompt to start up ECL^{i}PS^e.[2]
   - Compile and load the program by issuing the command `[bratwurst].` at the command prompt (note the `.` at the end).
   - Type `schedule(EndTime).` to find a solution. 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^{i}PS^e will then run rather quietly and you won't see progress messages as it works.

   (a) [**5 points**] If kickoff is at noon, at what time should you start preparing your bratwurst so that you don't miss any of the game?

---

[1]However, ECL^{i}PS^e does in fact embed constraint programming in the general-purpose Prolog language. We'll study Prolog later in the course, but if you want to try learning it early and use it for this assignment, be our guest. You could start with *Prolog Programming in Depth* and/or the SWI Prolog documentation (which has a bunch of tutorial links at the start).

[2]Type `halt.` to close the command line interface.

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 homework. 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^{i}PS^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^{i}PS^e examples page, http://www.eclipseclp.org/examples; there's even a section on planning and scheduling. Feel free to discuss the ECL^{i}PS^e website's examples on Piazza!

2. [**50 points**] Now that you've actually used ECL^{i}PS^e, here are a few warm-up problems to get you started writing your own code. Pick **any five (5)** of these five math problems, and write ECL^{i}PS^e code to solve them. For each problem you picked, write your solution in the `solve` method of `problemName.ecl` file. Use the problem names specified in the handout.

For example, if you are solving the *nine triangles* problem, write a `nineTriangles.ecl` file such that you can run

```
$ eclps
$ [eclipse 1]: [nineTriangles].
...
nineTriangles.ecl compiled xxxx bytes in xxxx.xx seconds

Yes (xxxx.xx seconds)
$ [eclipse 2]: solve(MySolution).
...
MySolution = ...

Yes (xxx.xx seconds)
$ [eclipse 3]:
```

We have provided a script `minipuzzle` to simulate how the autograder will run your scripts. To use `minipuzzle`, place your solutions inside `/usr/local/data/cs432/cp/src` and run:

```
$ ./minipuzzle nameOfPuzzle
```

`minipuzzle` uses a utility function defined in `/usr/local/data/cs432/cp/utils/pprint` to pretty-print the solution. If there are multiple solutions to the problem, `minipuzzle` will print one solution per line. Note that `minipuzzle` will print the solution only, so we strongly recommend not to use it for development, but only to verify that your solutions match the expected format.

Problem 1: nine triangles   [10 points]
Consider the 9 triangles in the illustration (labeled a through i):
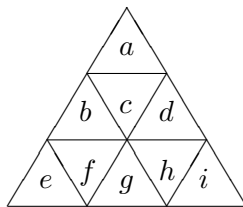
The numbers 1 through 9 can be arranged in the triangles such that the numbers in each of the 2×2 triangles sum to the same value n, that is:

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

3

For what values of n is there a solution to this puzzle?[3]

Expected output: `MySolution = [N1, N2, ...].`

Problem 2: concatenation   **[10 points]**

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 $6 \cdot (x||y) = (y||x)$.[4]

Expected output: `MySolution = [[X1, Y1], [X2, Y2], ...].`

Note: 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^{i}PS^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 . . .

Problem 3: isosceles triangles   **[10 points]**

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.[5]

Expected output: `MySolution = [[Base1, Side1, Side2], [Base2, Side2, Side2], ...].`

Problem 4: fun five   **[10 points]**

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?[6]

Expected output: `MySolution = N.`

Problem 5: small square   **[10 points]**

Find the smallest square number (perfect square) that uses each digit (i.e., $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$) once and only once.[7]

Expected output: `MySolution = N.`

Problem 6: equations   **[10 points]**

Find all integer tuples `[a,b,c,d,e]` that solve the following system of integer linear equations:

$$a^2 = a + b - 2c + 2d + e - 8,$$
$$b^2 = -a - 2b - c + 2d + 2e - 6,$$
$$c^2 = 3a + 2b + c + 2d + 2e - 31,$$
$$d^2 = 2a + b + c + 2d + 2e - 2,$$
$$e^2 = a + 2b + 3c + 2d + e - 8.$$

The search space is infinite—include a short explanation of how you know you have found *all* the solutions (this may involve a tiny bit of math).[8]

[3]From USAMTS '04–'05, Round 2, Question 1. www.usamts.org.

[4]From USAMTS '04–'05, Round 3, Question 1 (a). www.usamts.org.

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

[6]White House Kids Math Challenges (formerly www.whitehouse.gov/kids/math/), designed by David Rock of UMass-Dartmouth and Doug Brumbaugh of UCF.

[7]Ibid.

[8]From USAMTS '11–'12, Round 1, Question 2. www.usamts.org.

Expected output: `MySolution = [[A1, B1, C1, D1, E1], [A2, B2, C2, D2, E2], ...].`

Hints:

- If you write a function `solve(X)` that constrains `X` to be a solution of some problem, you can type `solve(X).` at the ECL^{i}PS^e prompt to find a solution, and repeatedly press ';' to find more solutions. You can alternatively type `findall(X, solve(X), List).` to set `List` to a list of all solutions, or `findall(X, solve(X), List), length(L, N).` to also set `N` to the number of solutions.

- If there are a lot of solutions, as in problem 2, then `List` will be too long and ECL^{i}PS^e will only print the beginning of it followed by '...'. Here are some helpful ways you could print all solutions (any one of these 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 ECL^{i}PS^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 ECL^{i}PS^e will print it in full without special coaxing.

  - `solve(X), print(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`)! This approach has the advantage that it prints the solutions as it finds them, without first constructing a `List` of all of them. We'll learn more about `fail` when we study Prolog later. 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 ECL^{i}PS^e to a file. You can then use any tools you like to analyze the output (e.g., `python`, or Unix commands like `sort`, `uniq`, and `wc`.)

Resource Constrained Project Scheduling (RCPS)

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 ECL^{i}PS^e to solve some of these problems.

Their description of the problem is in `rcps/rcps.pdf` along with the necessary data. Read through it, paying particular attention to the description of the data format and to problems 1-3.[9]

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.

3. **[0 points]** You don't need to turn anything in for this question; just read `rcps.pdf`, which is included in the handout directory.

---

[9]You may also want to take a look around their website, which is mirrored at http://cs.jhu.edu/~jason/432/hw-cp/benchmrx/rcps.html; the original URL was http://www.neosoft.com/~benchmrx/rcps.html.

4. [**25 points**] Solve the first problem from the benchmark set using ECL^{i}PS^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 ECL^{i}PS^e format.

For this problem, you need to consider only the precedence constraints in section 2 (and the time constraints in section 1); you may ignore the labor and zone constraints. A reference solution is provided in `rcps_s1.data`.

(a) [**20 points**] Turn in your ECL^{i}PS^e code, named `problem1.ecl`, as well as your list of task **start times**. If you wrote a script to generate the ECL^{i}PS^e code, make sure that script is included in your submission too. Your list of start times should be in a separate file called `problem1.solution`. Make sure your output is in the correct format.

You should encode each start time as an integer representing the number of work minutes since the overall manufacturing process began. For example, your output file should begin like this (the numbers are made up):

```
Found a solution with cost 600
asm_1.step_1 324
asm_1.step_2 131
asm_1.step_3 111
...
```

If you prefer, you can also include end times as in the `bratwurst` example:

```
Found a solution with cost 600
asm_1.step_1 324 - 356
asm_1.step_2 131 - 133
asm_1.step_3 111 - 400
...
```

(b) [**5 points**] Include in your writeup a description of what you did, as well as the total time your ordering requires to complete all the tasks.

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* homework, 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 each time as an integer representing the number of work minutes since the manufacturing process began. For example, `11/2+03:25` becomes simply `9655` [$= 25 + 3*60 + 450 + 10*2*450$]). You'll use this simpler format throughout your solution and in your final schedule.

5. [**0 points**] Write ECL^{i}PS^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, ECL^{i}PS^e will be far too slow to solve it. We'll deal with this in the next section.

6. **[45 points]** Why is your program so slow? Isn't this what ECL^{i}PS^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 ECL^{i}PS^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 ECL^{i}PS^e to do this? The easiest way is this hack: `minimize` prints the costs $v_1, v_2, \ldots$ of the intermediate solutions it finds.[10] 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 ECL^{i}PS^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.

When ECL^{i}PS^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` $< v_i$). We saw the same problem in the graph-coloring animations in class.

If you were trying to schedule these tasks by hand, without using ECL^{i}PS^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, ECL^{i}PS^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`.

---

[10]As long as you use ECL^{i}PS^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.

- `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. Answer the following questions, even if you did not ultimately find a solution:

(a) [**5 points**] Discuss how well your solution did by answering in your writeup:
   - What is the lowest-total-time schedule that you can find that satisfies all the constraints?
   - 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 the `ugrad` or `grad` servers? Which machine did you use? (approximate is okay if it's really long)

(b) [**15 points**] Describe clearly how your search method works, and why you thought it would be effective.

(c) [**5 points**] 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?

(d) [**20 points**] Make sure your code is in `problem2.ecl` and your best ordering is in `problem2.solution` following the format described in problem 4. Also include additional scripts you may have used to generate your ECL^{i}PS^e file. Please submit even if you did not find a solution.

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, you can still answer most of the questions and get partial credit.

7. [**Extra credit**] [**variable points**] 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-ec.ecl` and `problem2-ec.solution`, and describe your method and results in your writeup just as in the previous problem.

So how can you write your own custom heuristics? The standard heuristics supplied to you by ECL^{i}PS^e are actually just the names of ECL^{i}PS^e 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 ECL^{i}PS^e 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!