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

Spring 2005
Prof. J. Eisner
TA: John Blatz

Due date: Wednesday, March 23, 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: As always, the work you hand in should be your own, in accordance with University regulations at `http://cs.jhu.edu/integrity.html`

How to hand in your work: Specific instructions will be announced before the due date. Your programs have to run in ECL^{i}PS^e, which is installed on `barley` and both the CS undergrad and grad networks. 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 `barley`.

In this assignment, 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 assignment, we're only going to ask you to work using a very simple subset of ECL^{i}PS^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 at the ECL^{i}PS^e website, `www.icparc.ic.ac.uk/eclipse`, 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.

   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.

   I have provided an ECL^{i}PS^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 ECL^{i}PS^{e}by doing the following:

(a) Logged on to `barley`, type "`eclipse`" at a command prompt to start up ECL^{i}PS^e.

(b) Execute the command "`compile('bratwurst.ecl').`"

(c) Evaluate the function "`schedule(EndTime).`". This will return 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 command `eclipse -b bratwurst.ecl -e 'schedule(EndTime)'`, and you can redirect this output to a file or program of your choice in the usual way.

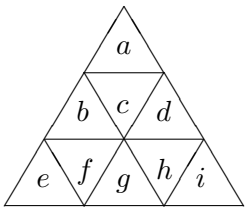
If kickoff is at noon, at what time should you start preparing your bratwurst so that you don't miss any of the game? **Report the answer to this question in your `README`.**

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 in the next part. 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 at `http://www.icparc.ic.ac.uk/eclipse/examples/`.

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

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



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 $6(x||y) = (y||x)$.[2]

[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`.

(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]

(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 ECL^{i}PS^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`.

3. Barry Fox and Mark Ringer of the American Association for AI have put together a series of benchmark problems in planning and scheduling, in order to facilitate comparison between different algorithms for solving this type of problem. In this section, we will use ECL^{i}PS^e to solve some of them.

   Their description of the problem is on `barley` 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 go to their website, `http://www.neosoft.com/~benchmrx/rcps.html` and take a look around. The data files are already on `barley`, 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 part, just read `rcps.pdf`.

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

[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.

4. 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; you may ignore the labor and zone constraints.

   A reference solution is provided in `rcps_s1.data`. **Turn in your ECL^{i}PS^e code, named `problem1.ecl`, as well as a list of task start times in the format of `rcps_s1.data`. This should be in a separate file called `problem1.solution`. 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 ECL^{i}PS^e code, **turn in that script as well**.

5. Write ECL^{i}PS^e code to solve the second problem from the benchmark set. For this part, 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.

   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 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 dues this using the *branch and bound* algorithm: first, it finds any labeling of `AllVars`, and notes the maximum value of `EndTime` consistent with this labeling. It then adds a new constraint that states that `EndTime` must be strictly less than this value, and tries to find another labeling with the added constraint. If it can do so, it constrains `EndTime` to be less than this maximum value, and iterates this procedure 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 strong

tip-off that this algorithm is not 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 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. The easiest way for you to do this in ECL^{i}PS^e is to just hack it; `minimize` prints the cost of intermediate solutions that it iterates through, so just stop execution at some point, then run the program again, adding the additional constraint that the cost should equal the cost of the last iteration, and tell ECL^{i}PS^e to simply find and print out a labeling rather than to minimize it.

Unfortunately, this is not good enough; even without the minimization, just the command `labeling(AllVars)` will 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 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. If the first couple variables are ill-selected, you can see how this could take a while.

Fortunately, ECL^{i}PS^e provides a constraint that gives you some control over the way that labeling is done. For our purposes, this command 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.icparc.ic.ac.uk/eclipse/doc/bips/lib/ic/search-6.html`.

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 ECL^{i}PS^e program for this part, which you should call `problem2.ecl`. Include in your `README` a description of what constraints you used, what parameters you used for `search`, and the best total time you could find. 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`.

7. [**Extra credit**] You may have noticed in the documentation for `search` that you are allowed to create custom variable selection and value choice methods. See if you can come up with 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.ecl` and `problem2.solution`, and describe your method and results in your `README` just as in the previous problem.

Notes on ECL^{i}PS^e: The time format 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.

You are allowed to interrupt work at the end of a shift and pick up where you left off at the start of the next shift, and also there is no difference between first and second shift, so you probably want to encode times as an integer representing the number of minutes since the start time (e.g. `11/2+03:25` becomes simply `10555` [= 25 + 3*60 + 450 + 11*2*450]).

Documentation for all of ECL^{i}PS^e's predicates is available at `http://www.icparc.ic.ac.uk/eclipse/doc/bips/fullindex.html`. You may find this helpful in writing your programs.