600.432/632 — Declarative Methods
Homework 4: Mixed Integer Programming

Prof. Jason Eisner — Spring 2026

Due date: Friday, April 10, 2:00 PM

This homework is about Mixed Integer Programming. You will be using the ZIMPL modeling language and the SCIP solver.

Instructions

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

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

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

- **Homework 4 Written**: Submit a PDF with your solutions to each problem. After submitting, please mark the area for each question using Gradescope's interface.

- **Homework 4 Code**: Submit `knapsack.zpl`, `events.txt`, `homeworks.txt`, `schedule.zpl`, `README`, `sudoku1.zpl`, `sudoku2.zpl`.

You are allowed unlimited submissions, but the problems will not be graded until after the deadline. Please try submitting early (even if it is incomplete) to ensure your output formats are correct.

Getting familiar with ZIMPL

1. [**8 points**] Log onto one of the `ugrad` machines and look at the example ZIMPL programs in the handout directory. The short `transport` programs were shown on slides in class, and show successively better ways of expressing the same small problem in ZIMPL. You can find the ZIMPL manual at http://zimpl.zib.de/download/zimpl.pdf, and I recommend looking through it as you do this homework.

   Now let's try SCIP.[1] Type `scip` (or `rlwrap scip`) to start up the *interactive interface*[2] of the SCIP solver. Type `help` to see what commands are available in the interactive interface.

   Let's solve the `transport4` problem as follows. First type `read transport4.zpl` to load the problem. To see the variables and constraints in their internal form (translated from ZIMPL), type `display problem` (or just `di prob`[3]). Next, type `optimize` (or `opt`) to run the solver. The solver will show its progress as it proceeds.[4]

---

[1]It is installed on the `ugrad` and `grad` machines. If you want to install SCIP on your own machine, it is at https://www.scipopt.org/index.php#download. Just downloading a binary executable file is probably easiest. Any version compiled with ZIMPL support should be okay. The version installed on `ugrad` and `grad` is compiled with CLP to use as its LP solver, but other choices of solver ought to work too.

[2]One can also call SCIP as a library directly from C++.

[3]Commands in SCIP can be abbreviated to unambiguous prefixes.

[4]On a big problem when the solver is taking too long, you can hit Ctrl-C to stop it partway through. (Or you can type `presolve` instead of `optimize` to tell it to stop itself after the presolver stage.) You can then use the `display` commands to look at the solver statistics and the best solution so far (if any). To *continue* after Ctrl-C, type `opt` again; or to *start over*, type `newstart` and then `opt`, which will restart the solver but will use any conflict clauses already learned.

You probably now want to look at the solution. Type `display solution` (or just `di so`) to see the objective value and the optimal solution; note that variables assigned a value of 0 are not listed. For each variable that contributes to the linear objective function, its coefficient in the objective function is also shown.

Answer in your writeup:

(a) [**1 point**] What is the minimum transportation cost that can be achieved?

(b) [**1 point**] In this optimal solution, which producer is not operating at full capacity?

(c) [**1 point**] Which consumers are using multiple suppliers to meet their demand?

(d) [**1 point**] What is the most expensive (cost per unit) transportation route that is actually getting used?

Type `display statistics` to learn more about what the SCIP solver did on this problem. You can see from all of the zeros that SCIP didn't have to do much of anything! Since this is just a plain linear programming (LP) problem, SCIP merely had to call the underlying LP solver (namely CLP) a single time to solve it.

Note: If you want to do the above sequence of commands without going through the interactive interface, just type `scip -f transport4.zpl` at the Unix command line.[5]

Answer in your writeup by studying the "LP" section of the statistics:

(e) [**1 point**] How many iterations (i.e., iterations of the simplex method) did the LP solver need to solve this problem?

You can use other `display` commands to see more details of what the SCIP solver did with this problem, as well as the many settings that controlled the solver's behavior. Type `display` to see what those commands are.[6]

For example, `display transproblem` shows a preprocessed version of the problem that was used internally. You will notice that this uses a different set of variables (whose names start with `t_`, for "temporary"), and that the variables' domains have been reduced (by constraint propagation or by the presolver).

Answer in your writeup:

(f) [**1 point**] The reduced domains indicate that Alice will send at most 400 units to consumer 2, while Bob will send at most 300 units. Explain how these limits can be easily deduced from the problem in `transport4.zpl`.

(g) [**2 points**] Finally, what happens if you allow negative values? By default, variables are constrained to be `>= 0`. Try modifying the line

```
var send[Producer*Consumer];
```

to

```
var send[Producer*Consumer] >= -10000;
```

Discuss the solution that is found in this case.

---

[5]Or you can also specify a different sequence of interactive commands via the command line, e.g.,
`scip -c 'set display verblevel 0' -c 'read transport4.zpl' -c opt -c 'display solution' -c quit`

[6]Type `set` to see how to change the settings. Type `write` to see what you can write to a file in a format for other automatic programs to read.

Programming in ZIMPL

2. [**27 points**] Now let's try a little simple ZIMPL programming. We'll address the so-called "KNAPSACK problem." You are a burglar in an empty house, and you have to decide how to fill your knapsack. It only holds 80 pounds, so you can't take everything in the house. Which items should you take?

- You could take the most valuable items. But some of them (like the piano) are too heavy.

- You could take the lightest items. But some of them (like the stale marshmallow under the table) are not very valuable.

- You could focus on the items that have the greatest value per pound, starting with the jewelry. This is a good compromise, but still not quite right. You might prefer 20-pound items over 21-pound items because you can fit 4 of them into your 80-pound knapsack rather than 3 ... even if the 21-pound ones happen to be slightly more valuable per pound.

Each item has a value and a weight. The point is that there are interesting tradeoffs here. Taking some items may block you from taking others. So we have a combinatorial optimization problem, of the sort covered in this class.

It turns out that if all of the weights are small *integers* (e.g., from 0 to 80), then there is a fast dynamic programming algorithm for the KNAPSACK problem. But in the general case, the problem is NP-hard to solve exactly.

Let's try using integer linear programming. SCIP doesn't know anything about dynamic programming, but at least it can attack KNAPSACK by using its usual generic branch-and-cut strategies. This question examines whether that approach is successful on this problem.

(a) [**0 points**] Read the `knapsack.zpl` file. Complete the ZIMPL program by filling in the missing objective function and the missing constraint. This should be easy.

(b) [**1 point**] Answer in your writeup: How fast does this program (under SCIP) solve the 10,000-item problem provided in `knapsack.txt`? (Note that the filename `knapsack.txt` is hardcoded into the program.) Again, this is being done without dynamic programming.

(c) [**2 points**] You can use `display solution` to see the value of the objective function, followed by the very long optimal assignment. What does the objective function represent? What does the optimal assignment represent?

(d) [**4 points**] Save the solution you just found to a file by typing `write solution sol1`. Now change the declaration
`var take[I] binary;`
to
`var take[I] <= 1;`
Solve this changed problem, which is a *pure LP problem with no integrality requirements*, and do `write solution sol2`.

Now at the Unix command line, type `comm -3 sol1 sol2`. For sorted files, this lists all the lines that appear only in `sol1`, and all that appear only in `sol2`, with the latter indented.

  i. Study the differences, and describe in English what has changed between the two solutions. How is `sol2` packing the knapsack differently and why?

  ii. How many of the 10,000 `take` variables turned out to be integers in `sol2` even though they weren't required to be?

  iii. Can you give an intuition as to why this is? (*Hint:* What would you expect about the value-to-weight ratio of the taken items in the non-integer case?)

  iv. How would you expect this property of `sol2` to affect the efficiency of `sol1`?

When you are done answering the above questions in your writeup, change the program back to a mixed integer program.

(e) [**4 points**] You would like to know how many items are in the optimally packed knapsack, what their total value is, and how much room is left in the knapsack. One way to do it would be to write a program to analyze SCIP's output on `sol1`. However, there is an easier way.

Introduce variables `count`, `totalvalue`, and `spareweight` into your ZIMPL program and constrain them so that they answer the three questions above, respectively. Note that by default, variables are constrained to be `>= 0`. This should allow you to simplify your code.

Now solve the problem again (using `optimize`). What are the values of those three variables? To find out without printing the whole assignment, type `display value count`, `display value totalvalue`, and `display value spareweight`.

Note: It may be easier here and below to type in all the commands on a single line:

```
read knapsack-solution.zpl opt disp val count disp val totalvalue disp val spareweight
```

(f) [**2 points**] Does the solver do the same thing every time? Solve the problem again *without* changing the program. You can either start SCIP again, or use the `read` command to reread the problem, or use the `newstart` command. Then you can type `opt` again.

Were the solver statistics (`display statistics`) the same as last time? Why or why not?

Were `count`, `totalvalue`, and `spareweight` the same? Why or why not?

(g) [**3 points**] It will be easier to unpack the knapsack when you get home if it contains fewer items. Change the linear objective function so that it encourages a large `totalvalue` (as before) but also encourages a small `count`.

What is your new objective function? How does this affect the values of `count`, `totalvalue`, and `spareweight` in the optimal solution? Why?

(h) [**3 points**] Argue that the previous problem is actually just solving a different KNAPSACK problem. That is, explain how your fancy revised problem that encourages fewer items could be reduced to an plain old KNAPSACK problem and solved with an plain old KNAPSACK solver.

(i) [**4 points**] Let's place one additional constraint on the program, so that it is no longer a plain old KNAPSACK problem. Some items are radioactive. For safety, you must ensure that the total radioactivity of the knapsack is `<= 20`. The radioactivity of each item is given in the fourth column of `knapsack.txt`.

(Since the radioactivity is a real number rather than an integer, a dynamic programming approach wouldn't work at all here.)

Add this constraint to the ZIMPL program. So how fast is SCIP on this problem? How does the addition of the radioactivity constraint change `count`, `totalvalue`, and `spareweight`? Why?

(j) [**4 points**] Submit your final program as `knapsack.zpl`.

(k) [**extra credit**] Study the solver output and try to figure out where the solver is spending its time on this problem. You might be particularly interested in the output of `optimize`, `disp displaycols`, `disp stats`, and `disp transproblem`. You could comment on things like the shape of the branch-and-bound tree, the total number of LP (simplex) iterations, the effective presolving and propagation strategies, etc.

3. [**21 points**] Here's a fairly typical kind of MILP problem, so I'll let you work out the details. (A version of this problem appeared on a previous Declarative Methods exam.)

It's the end of term and you have too many things to do! You decide to use a MILP solver to plan your time. Here's what's on your mind as you model the problem:

(a) [**0 points**] You decide to plan your time for the next week. *Hint:* start with by defining a set

```
set Day := {1..7};
```

and assume that each day `<i> in Day` consists of 24.0 hours.

(b) [**0 points**] For *each* day, you need to decide how many hours to spend on each of work, sleep, and play. (There is no reason to require these numbers to be integers.)

(c) [**4 points**] You need sleep. In particular, in any 3-day period, you must get at least 18 hours of sleep. (In your writeup, explain how you interpreted this constraint for 3-day periods that are only partially contained within the week. There is more than one reasonable approach.)

(d) [**4 points**] Subject to this hard constraint, your objective is to maximize the amount of Fun you have. Ordinarily, every hour of play will give you `base_fun_rate` points of Fun:

```
param base_fun_rate := 1;
```

However, each day, there exist various special events (such as parties or movie screenings) that can supply you with Fun at a higher rate. If you attend an event, you must attend the entire event.

You should make a short file `events.txt` with one row for each special event, listing the relevant information about the event. *Hint:* Each special event has a string name, takes place on a particular day, consumes a certain amount of play time, and provides you with a certain rate of Fun per hour. What would the solver do if an event's fun rate is lower than `base_fun_rate`?

(e) [**4 points**] Unfortunately, you also have homework. You should make a short file `homeworks.txt` containing the relevant information about each homework. Each homework has a string name, is due on a particular day, requires a particular number of hours to complete, and has a particular penalty rate.

If an homework is due on day `i`, then its hours can be spread out arbitrarily over the days $1, 2, \ldots, i$ (but not the days later than `i`!).

There's not a hard constraint to do all of the homework. If an homework requires 10 hours but you spend only 8 hours total on it, then you'll just get a lower grade—but this isn't Fun. For example, if the penalty is 3 points of Fun per skipped hour, your penalty for skipping the last 2 hours of this homework will be $3 \times 2 = 6$ Fun points.

(Notice that you might end up with a penalty on day 4 because you chose to party rather than work on day 2.)

Suppose some parts of the homework are worth more than others. Can you still encode this situation using the approach above, or do you need a more complicated model?

(f) [**5 points**] You are a less efficient worker when you get less sleep. We say that you have a *sleep deficit* on day `i` if you have had less than 24 hours of sleep total over the 3-day period $i-2, i-1, i$. (Again, in your writeup, explain how you interpreted this constraint for 3-day periods that are only partially contained within the week.)

On days when you have a sleep deficit, your work is only 75% as efficient as usual. For example, it takes you 8 hours to do 6 hours' worth of work.[7]

(The 75% number should probably be a named parameter, in case you want to play with it.)

(g) [**4 points**] Hand in your commented ZIMPL model, `schedule.zpl`, along with data files `events.txt` and `homeworks.txt` that contain the parameters of an interesting scheduling problem. In addition, hand in a `README` which states how to run the solver from the command line. **Important:** Your data files should result in a non-trivial solution. In other words, the *optimal schedule for your inputs* should illustrate the tradeoffs in the situation. This schedule says you sometimes get enough sleep (no deficit) but not always; you complete some homeworks but not all

---

[7]So if an homework requires 7 hours, then it's not enough to spend 7 hours on it if you have a sleep deficit during some of those hours. In fact, even if you spend 8 hours, all on sleep deficit days, then you skipped the last 2 hours of the homework and will be penalized as above.

of them; you do some of your work before the day it's due but not all of it; and you go to a few special events but not all of them. Just like life! (There are no *constraints* that explicitly require this kind of non-trivial solution; I'm just saying you should find sample *parameters* that happen to elicit such a solution.)

(h) [**2 points**] In your writeup, discuss the solution that you found.

(i) [**Extra credit**] [**variable points**] Add a notion of late days. For example, you have a total of k late days that you can allocate over all the homeworks. Submit a modified version of your `schedule.zpl` with discussion.

(j) [**Extra credit**] [**variable points**] Instead of having a binary notion of "sleep deficit," devise a more plausible function to describe how your work efficacy falls off nonlinearly with as the amount of sleep you've gotten recently. Submit a modified version of your `schedule.zpl` with discussion.

(k) [**Extra credit**] [**variable points**] Instead of considering just a week, plan your time for the next year (365 days). Essentially the same model `schedule.zpl`, but you will have to construct longer input files to describe the events and homeworks, for example by using a script with a random number generator. In your `README` (**not the writeup**), state how to generate these files and describe what assumptions you made. Do you see any evidence of a phase transition in runtime (as with random 3-SAT) as the problem goes from being underconstrained to overconstrained?

4. [**19 points**] You may know about Sudoku puzzles (if not, look online to make sure you understand the rules!). The idea is to fill the cells of a 9×9 grid with the digits 1–9, subject to the following constraints:

- In each row, each of the 9 digits appears exactly once.
- In each column, each of the 9 digits appears exactly once.
- In each of the nine major "blocks" (3×3 squares), each of the 9 digits appears exactly once.

Usually a few of the numbers are given, i.e., a few cells are constrained to have particular values. Then you have to fill in the rest of the cells consistent with the above constraints. Many people find this addictive.

It is pathetically easy to solve Sudoku puzzles published for humans by using constraint programming approaches (although Sudoku is NP-hard in general, and there is some interesting work on designing powerful propagators for the general problem). So we'll give the solver some harder challenges below.

For generality, let's deal with $n^2 \times n^2$ puzzles whose cells must be filled with the numbers 1 through n^2; taking $n = 3$ gives the usual case.

(a) [**5 points**] You can find an incomplete ZIMPL program in `sudoku.zpl`. Finish it. How long does SCIP take to solve the "very hard" problem in `sudoku.txt`?

To view the result, pipe the output of SCIP through the `sudoku-decode` program that is provided. That is, at the Unix command line, write

```
scip -f sudoku.zpl | ./sudoku-decode
```

Submit your program as `sudoku1.zpl`.

(b) [**1 point**] Include the decoded solution in your writeup, checking that it does indeed satisfy the requirements.

Now let's have a little fun by constraining the solution not with some given numbers, but instead in other entertaining ways.

Copy your program to `sudoku2.zpl`. Delete the `givens` constraint that forces the assignment to respect the givens.

Instead, add a `rotsymm` constraint that forces the assignment to be 180° rotationally symmetric. (For example, the lower right corner should have the same digit as the top left corner, and in fact the bottom row should just be the top row backwards.)

The solver should discover that the problem is infeasible (that is, UNSAT).

(c) [**1 point**] In your writeup, give a simple argument in English that demonstrates that no valid 9×9 sudoku can be 180° rotationally symmetric. (*Hint:* Think about the constraints being placed on the central 3×3 block.)

(d) [**4 points**] The previous argument doesn't apply to the $n = 4$ case, however, since none of the sixteen major 4×4 blocks of a 16×16 grid is in the center.

So is it possible to have a rotationally symmetric 16×16 sudoku? Use SCIP to find out! If yes, include the decoded solution in your writeup. If it is still UNSAT, give a different argument in English that explains why.

Either way, hand in the program as `sudoku2.zpl`. How long did SCIP take to finish?

(e) [**4 points**] Even though it's not possible to have a rotationally symmetric 9×9 grid, use SCIP to find one that is *as close* to rotationally symmetric as possible. Specifically, you should try to minimize the number of cells that are not equal to their rotational counterparts.[8] How close can you get to rotationally symmetric?

(*Hint:* You may find `vabs` to be helpful here (see section 4.13 of the ZIMPL manual).)

(f) [**2 points**] Your solution to the previous problem is only one of a family of symmetric solutions. For example, if you changed all the 5's to 8's and vice-versa, you would still have a valid Sudoku that is as close to rotationally symmetric as before. In fact there are $n!$ different ways to permute the digit values, none of which change the quality of the solution.

Add a symmetry-breaking constraint `no_permute_digits` that blocks these permutation symmetries. Explain your strategy in your writeup, and report whether adding this constraint speeds up SCIP or slows it down (in principle either is possible). Hand in your program as `sudoku3.zpl`.

(*Hint:* Think about getting the first row into a standard form.)

(g) [**Extra credit**] [**variable points**] There are other symmetries that one ought to break. For example, permuting the rows or columns in certain ways is guaranteed to preserve the validity of the Sudoku and the amount of rotational symmetry. Add constraints to break these symmetries as well. Hand in your program as `sudoku3_ec.zpl` and discuss in your writeup.

(h) [**2 points**] Now go back to $n = 3$. Suppose you allow the `x` variables to be real numbers in the range $[0, 1]$, rather than requiring them to be integers. Now you *could* have a solution that satisfies all the constraints including `rotsymm`. What is the "simplest" such solution? (There is no need to change the program or run SCIP; just think about it!)

(i) [**Extra credit**] [**variable points**] Construct some other cool sudoku grids and show them off! For example, you could try to require each $n \times n$ block to be a magic square (so that each row and each column within the block has the same numeric sum). Or you could require additional "pretty" sets of n^2 cells—not just rows, columns, and blocks—to contain the numbers 1 through n^2 exactly once each.

5. [**15 points**] The point of this question is to give you some practice in thinking about the dual of an LP problem. Typically, the dual problem will have some interesting interpretation that is not so obviously related to the original problem. (E.g., the dual of max-flow can be interpreted as min-cut, and vice-versa.)

---

[8]You can get a solution that is prettier (in my opinion) if you instead try make each cell be *numerically* close to its rotational counterpart, so that if the cell is 3, then its rotational counterpart should best be 2 or 4 (rather than 9) if it can't be 3. Try it if you like—it's only a very small change!

Returning to the standard KNAPSACK problem (no radioactivity), consider the LP relaxation (i.e., drop the integrality constraints). This is a maximization problem with n variables $(x_1, \ldots x_n)$ and $n+1$ constraints (other than the implicit constraints $x_i \geq 0$). One constraint gives the capacity of the knapsack, and n constraints have the form $x_i \leq 1$ (since the original ILP problem had $x_i \in \{0, 1\}$).

(a) [**1 point**] Why can the relaxed LP problem get a higher value of the objective than the original LP problem? Specifically, how does it make your job easier as a robber with a physical knapsack?

(b) [**4 points**] Derive the dual LP problem. Since duality swaps constraints and variables, and swaps max and min, this should be a minimization problem with $n+1$ dual variables $(y_0, y_1, \ldots y_n)$ and n dual constraints. Starting with the primal problem, you should be able to mechanically write out the dual constraints and dual objective.

Write your answer in mathematical notation or in ZIMPL. (If you write it in ZIMPL, you can actually try it.)

(c) [**1 point**] What can you say about the minimum value of the dual objective? (Will it equal the weight, or the value, of the optimally filled knapsack? Higher? Lower?)

(d) [**2 points**] What interpretation can you give to dual variable y_0?

Hint: Note that y_0 is the shadow price for the capacity constraint, in other words, the derivative of the primal objective with respect to the capacity. Figure out what this means in terms of the knapsack.

Further hint: The following terms may help you phrase your answer. Define the *density* of each item to be its value per pound. Given the optimal solution to the LP relaxation, there is some item that you'd most like to add "next" (or continue adding if it is already partly added) if the capacity were to increase slightly. Call this the *borderline item*.

(e) [**2 points**] What interpretation can you give to dual variables $y_1, \ldots y_n$?

Hint: Again, it helps to start by regarding these as shadow prices, and then translating that idea to the physical setting of the knapsack problem. For y_i, what increases slightly is not the knapsack capacity, but rather something (what?) that is related to item i.

(f) [**2 points**] Suppose for the sake of argument that y_0 is **fixed** to some constant. Now consider dual variable y_i where $1 \leq i \leq n$. There will now be two constraints on y_i, including the implicit constraint $y_i \geq 0$. When the dual objective is minimized with respect to these constraints, what value will y_i take on (as a function of y_0)? For which values of y_i will $y_i > 0$?

(g) [**3 points**] The above question shows that once y_0 is chosen, the values for the other y_i will be fully determined. The remaining question is how y_0 will be chosen.

But you've already characterized the optimal value of y_0 in your answer to question 5d.

What about the dual problem forces y_0 to this optimal value in the dual solution? Specifically, what prevents y_0 from being too big? What prevents it from being too small?

What would these too-big and too-small values of y_0 correspond to in the primal setting?

Note: Originally, instead of Sudoku, this homework's main problem was going to be about graph drawing—choosing positions in the (x, y) plane for the vertices and edges of a directed graph so that the edges tend to (1) be short, (2) point downward, (3) avoid crossing other edges, (4) avoid bending too much. This combines several NP-hard challenges into a single mixed integer programming (MIP) problem. Other aesthetic criteria can easily be incorporated into the program. This is a harder but very satisfying problem that really exercises the MIP solver, with binary, integer, and real variables, and ought to produce cutting-edge results in graph drawing (though not necessarily fast). Unfortunately, that would require finding graph data and visualization tools. Maybe next year!