Modelling, Simulation & Optimisation (H9MSO)

2. Integer Programming

Today's Outline

- Integer Programming
- The n-Queens Problem
- A Warehouse Location Problem
- Logical Constraints
- The Travelling Salesman Problem
- Cutting Stock Problem

Integer Programming

- In LP we allow variables to take fractional values, even if this doesn't make sense for our application
  - we assume that rounding the values to the nearest integer will still give a feasible and almost-optimal solution.
  - Last week, we produced 81.82 objects
- In general we can't do this, because we might get a very sub-optimal solution or even an infeasible one.
- For many problems fractional values are useless, but instead of LP we can use *Integer Programming* (IP, sometimes written ILP where *L* stands for *Linear*).
- An IP is simply an LP with extra constraints stating that all the variables must take discrete values (usually integers).
- A problem containing both integer and continuous variables is a *Mixed Integer Program* (MIP or sometimes MILP).

- Although in principle adding constraints makes the problem "smaller" in some sense, IP and MIP are much *harder* to solve than LP.

- There are polynomial-time algorithms for LP but not for IP/MIP because they are NP-hard.

- (NP-hard problems are a large class of problems that have no polynomial time algorithms, as far as we know.) If anyone could find such an algorithm then the entire class could be solved quickly, and the inventor would be rich and famous.)

A Short Introduction to PuLP

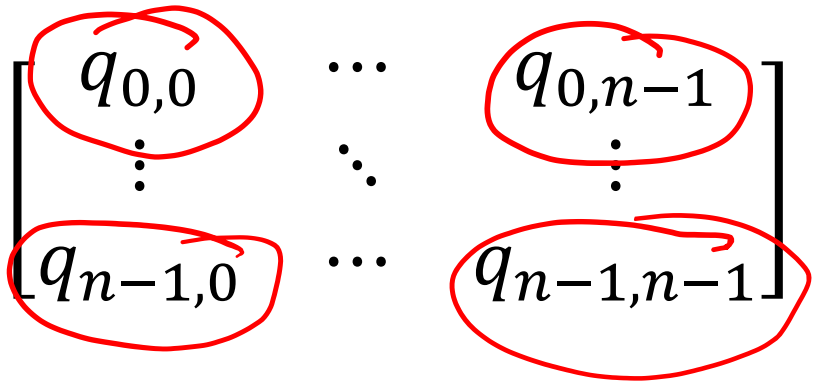
- PuLP makes full use of Python as programming language.
- Install PuLP with

```
pip3 install pulp
```

- Follow the Instructions on Moodle.

Example: n-Queens Problem

- Here's one approach. For each of the n x n squares on the board let's have a binary variable $q_{i,j}$ which is 0 if there's no queen on it, and 1 if there is a queen on it. Let's arrange them like this:



- In pulp we write:

```
q=pulp.LpVariable.dicts("Q",
                        (range(0,n), range(0,n)),
                       lowBound=0, upBound=1,
                       cat=pulp.LpInteger)
```

Example: n-Queens Problem

- We try to maximise the number of queens on the board:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} q_{i,j}$$

- In pulp we create the problem and then add the value to maximise or minimise:

```
prob = pulp.LpProblem("Queens",
                      pulp.LpMaximize)
prob+= pulp.lpSum([ q[i][j]
                    for i in range(0, n)
                      for j in range(0, n)])
```

List of formulas

Example: n-Queens Problem

- Constraints:
  - Each **row** $i = 0..n-1$ has exactly one queen:

$$\sum_{j=0}^{n-1} q_{i,j} = 1$$

  - In pulp we write this as n constraints that are added one after the other:

```
for i in range(0, n):
    prob += pulp.lpSum([q[i][j]
                        for j in range(0, n)]
                       ) == 1
```

Constraint is added

Example: n-Queens Problem

- Constraints:
  - Each **column** $j = 0..n-1$ has exactly one queen:

$$\sum_{i=0}^{n-1} q_{i,j} = 1$$

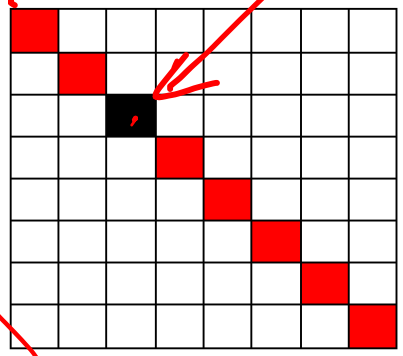
  - In pulp we write this as n constraints that are added one after the other:

```
for j in range(0, n):
    prob += pulp.lpSum([q[i][j]
                        for i in range(0, n)]
                       ) == 1
```

add Constraint

Example: n-Queens Problem

- Constraints:
  - And each diagonal has at most one queen:

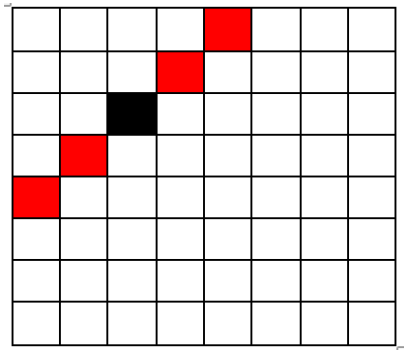


  - In pulp we write:

```
for i in range(0,n):
    for j in range(0,n):
        prob += pulp.lpSum(
            [ q[i+k][j+k]
              for k in range(-n, n)
                if 0<=min(i+k, j+k) and
                   max(i+k, j+k)<n  ]) <= 1,
```

Example: n-Queens Problem

- Constraints:
  - And each diagonal has at most one queen:



  - In pulp we write:

```
for i in range(0,n):
    for j in range(0,n):
        prob += pulp.lpSum(
            [ q[i+k][j-k]
                for k in range(-n, n)
                 if 0<=min(i+k, j-k) and
                    max(i+k, j-k)<n  ]) <= 1
```

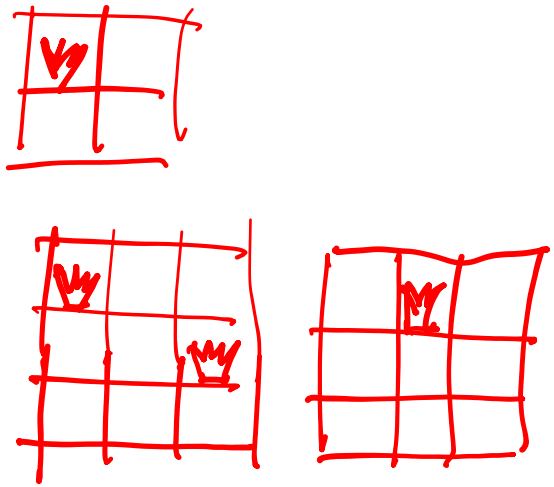
Example: n-Queens Problem

- To solve the problem we write in pulp: `status = prob.solve()`
  If `status==1` there is a solution, otherwise not.
- If we want to find all solutions, we run a loop that adds incrementally more constraints:

```
k=0
status=prob.solve()
while status==1:
    k+=1
    print_solution(n, q)
    prob += pulp.lpSum([ q[i][j]
                         for i in range(0, n)
                           for j in range(0, n)
                             if pulp.value(q[i][j])==1
                       ]) <= n-1
    status=prob.solve()
print(f'For n={n:d} there are {k:d} Solutions')
```

Example: n-Queens Problem

- For n=2 and n=3 there are no solutions.
- For n=4 there are two (symmetrical) solutions.
- For n=5 there are 10 solutions
- For n=6 there are 4 solutions
- For n=7 there are 40 solutions
- For n=8 there are 92 solutions
- For n=9 there are 352 solutions…



- Let's have a look.
  Please check the queens.ipynb file from Moodle.

Warehouse location problem

- A company considers opening warehouses at some candidate locations in order to supply its existing stores.
- Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply.
- Each store must be supplied by exactly one open warehouse.
- The supply cost to a store depends on the warehouse.
- The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized.

Warehouse location problem

- We can model this problem in a similar way to the LP examples, except that we now have integer variables.

- Define binary variables o_i

$$o_i = \begin{cases} 1; & if\ warehouse\ i\ is\ open \\ 0\ ; & otherwise \end{cases}$$

- and s_{ij}

$$s_{ij} = \begin{cases} 1; & if\ warehouse\ i\ supplies\ store\ j \\ 0\ ; & otherwise \end{cases}$$

- This is a typical use of binary variables to represent decisions.

Warehouse location problem

- Now the constraints.
  - We must ensure that each store is supplied by one warehouse:

$$\sum_i s_{ij} = 1$$

  - Each warehouse capacity c_i must be respected:

$$\sum_j s_{ij} \leq c_i$$

  - Of course if a warehouse is supplied then it must be open:

$$s_{ij} \leq o_i$$

Warehouse location problem

- **Note.** We could collect the last set of constraints into a smaller set using a modelling trick:

$$\sum_{j=1}^{S} s_{ij} \leq S o_i$$

5 instead of 100 variables.

- where S is the number of stores. This does the same as the other constraints, but it needs a bit of thought to see why. We'll return to this.
- The objective is to minimise

$$Z = k \sum_i o_i + \sum_i \sum_j k_i s_{ij}$$

- where k_i is the cost of supplying store j from warehouse i, and k is the warehouse fixed maintenance cost.

Logical constraints

- As we saw in the example, when we're modelling decisions by binary variables there are some modelling tricks that we need to know.

- For example, we might want to ensure that if one decision is made then at least one of two others is; or if several other decisions are made then so is another.

- Before we see more examples, here's a collection of these tricks for some set of binary variables $y_1 \ldots y_n$:

Logical constraints

- The decisions are mutually exclusive:

$$0 \leq \sum_i y_i \leq 1$$

$0 \leq y_i \leq 1$

- At most k of the decisions can be made:

$$\sum_i y_i \leq k$$

- At least k of the decisions can be made:

$$\sum_i y_i \geq k$$

- Exactly k of the decisions can be made:

$$\sum_i y_i = k$$

Logical constraints

- Now a less obvious one. If *any* of the y decisions are made, then so is another decision w:

$$\sum_{i=1}^{n} y_i \leq nw$$

$y_1 \vee \dots \vee y_4 \rightarrow w$

We multiply by n so that the constraint can be satisfied if all the $y_i = 1$.

- If *all* the y decisions are made, then so is w:

$$\sum_{i=1}^{n} y_i \leq n - 1 + w$$

$y_1 \wedge \dots \wedge y_4 \rightarrow w$

- If *at least* k of the y decisions are made then so is w:

$$\sum_{i=1}^{n} y_i \leq (k-1) + [n - (k-1)]w$$

Logical constraints

- What is the difference between

$$x + y \leq 2w$$

and

$$x + y \leq 1 + w\ ?$$

$x \vee y \Rightarrow w$

$x \wedge y \rightarrow w$

- The first means "if x or y is true then so is w" while the second means "if x and y are both true then so is w". (These are special cases of the above rules.)
- Consider:

$$x \leq y$$

$x \rightarrow y$

where x and y are binary, which describes the implication.
If $y = 0$ (decision y is not made), then x must be 0,
if $x = 1$ (decision x is made), then y must be 1.

Logical constraints

- We can represent the reverse of a decision y by $1 - y$.

- For example, if we take decision y_1 and we do not take decision y_2 then we take decision w:

$$y_1 + (1 - y_2) \leq 1 + w$$

$y_1 \wedge \overline{y_2} \rightarrow w$

- which is the same as:

$$y_1 - y_2 - w \leq 0$$

$\wedge$ conj.
$\vee$ disj
$\overline{x}$ neg
$\rightarrow$ impl

Example using logical constraints

- Suppose we want to solve the following puzzle:
  If it's raining then either I take an umbrella or wear a jacket, but not both. If I carry an umbrella then I can't enter a cafe. I want to enter a cafe. It's raining. What do I do?
- We'll use these 0/1 variables:
  - r is 1 if it's raining and 0 otherwise
  - u is 1 if I take an umbrella and 0 otherwise,
  - j is 1 if I wear a jacket and 0 otherwise,
  - c is 1 if I want to enter a cafe and 0 otherwise.

Example using logical constraints

- If it rains, then I carry an umbrella or wear a jacket:

  $$r \leq u + j.$$

  If $r = 1$ then the right-hand side must be at least 1, which forces at least one of u and j to be 1.
- I can't carry an umbrella and wear a jacket:

  $$u + j \leq 1.$$

  Both u and j can't be 1 because then the left-hand side would be 2.
- If I carry an umbrella, then I can't enter a cafe:

  $$u \leq (1 - c) \text{ or } u + c \leq 1$$
- I want to enter a cafe: $c = 1$
- It's raining: $r = 1$

Example using logical constraints

- The solution is $c = r = j = 1$ and $u = 0$:

  it's raining, I want to enter a cafe, I wear a jacket, and I don't carry an umbrella.

Another Example (big-M)

- Suppose that we have a real variable r and a binary variable b. If $b = 1$ then we want $r \leq 100$, otherwise we want $r \geq 100$.
- We can express this by two constraints:

$$r \leq 100 + M(1 - b) \qquad r \geq 100 - Mb$$

where M is some large number so that the red conditions are "uncritical". We can check that this does what we want it to.

- If $b = 0$ then these constraints reduce to:

$$r \leq 100 + M \qquad r \geq 100$$

- On the other hand, if $b = 1$ then the constraints reduce to

$$r \leq 100 \qquad r \geq 100 - M$$

- The purpose of the M was to "turn off" a constraint when b has some value.

domain constraint
$-200 \leq r \leq 200$
$M = 350$

Handling discrete values

- Here's another modelling technique. Suppose we have a variable x that we want to take discrete values that aren't consecutive integers, or maybe not even integers, for example x takes values from the set $\{1, 3, 4.5, 16\}$.

- To model this we can introduce new binary variables. In the example there are 4 discrete values for x so we define y_1, y_2, y_3, y_4.

- Then we use constraints

$$x = 1y_1 + 3y_2 + 4.5y_3 + 16y_4$$
$$y_1 + y_2 + y_3 + y_4 = 1$$

The travelling salesman problem (TSP)

- The TSP is a very well-known problem that has been attacked with all sorts of methods.
- We have a set of cities, or (more generally) nodes in a graph. Between each pair of nodes is an edge with a distance attached to it.
- In some cases the distance between city A and city B may be different than the distance from city B to city A: this is the *asymmetric* TSP.
- The problem is to plan a tour that visits each city exactly once, finishing up back where we started, while minimising the total distance travelled. (Not all the edges will be used in any tour.)

The travelling salesman problem (TSP)

- In preparation of the next G7 meeting, a secret messenger is sent from Berlin on a roundtrip to all the G7 capitals. What is the shortest roundtrip?

Capital	Country	latitude	longitude	Ottawa	Paris	Berlin	Rom	Tokyo	London	Washington
Ottawa	Canada	45.421106	-75.690308	0	5648	6127	6728	10319	5360	734
Paris	France	48.856697	2.351462	5648	0	876	1105	9715	343	6165
Berlin	Germany	52.517037	13.388860	6127	876	0	1183	8920	930	6710
Rom	Italy	41.893320	12.482932	6728	1105	1183	0	9859	1433	7217
Tokyo	Japan	35.682839	139.759455	10319	9715	8920	9859	0	9561	10902
London	UK	51.507322	-0.127647	5360	343	930	1433	9561	0	5898
Washington	USA	38.894985	-77.036571	734	6165	6710	7217	10902	5898	0

The travelling salesman problem (TSP)

- The ATSP is easily modelled as an IP. Here's a standard model found in 1954 by Dantzig, Fulkerson and Johnson.

- For each edge between nodes i, j (or trip between cities i, j) w define a binary variable x_{ij} .This is 1 if the trip is made, otherwise 0.

- In pulp we define the variables:

```
g7=pd.read_csv('G7 Distances.csv')
n=len(g7)
x=pulp.LpVariable.dicts("x", (range(0,n),range(0,n)),
                        lowBound=0, upBound=1,
                        cat=pulp.LpInteger)
```

$x_{ij} \in \{0, 1\}$

The travelling salesman problem (TSP)

- If the distance matrix is C_{ij} then we must minimize

$$Z = \sum_i \sum_j C_{ij} x_{ij}$$

Which we describe in pulp as:

```
prob = pulp.LpProblem("TSPG7",pulp.LpMinimize)
prob += pulp.lpSum([ distance(i,j)*x[i][j]
                          for i in range(0,n)
                            for j in range(0,n)
                              if i!=j
                        ])
```

with:

```
def distance(i, j):
    return g7.loc[i][capitals[j]]
```

The travelling salesman problem (TSP)

- Constraints:
  Each node i has exactly one successor (from every city we must travel to exactly one other):

$$\sum_j x_{ij} = 1$$

Which we describe in pulp as:

```
for i in range(0,n):
    prob += pulp.lpSum([ x[i][j]
                         for j in range(0,n)
                         if i!=j
                       ]) ==1
```

because $x_{i,i} = 0$

The travelling salesman problem (TSP)

- Constraints:
  Each node j has exactly one successor (from every city we must travel to exactly one other):

$$\sum_i x_{ij} = 1$$

Which we describe in pulp as:

```
for j in range(0,n):
    prob += pulp.lpSum([ x[i][j]
                         for i in range(0,n)
                         if i!=j
                        ]) ==1
```

The travelling salesman problem (TSP)

- The constraint that each node (city) must be visited exactly once is not easy to represent directly.
- The standard way is to prevent *subtours*, that is visits to *some* of the nodes that form a cycle. For every possible subtour, represented by a set S of edges, we add a constraint

$$\sum_{(i,j)\in S} x_{ij} \leq |S| - 1$$

- That is, not all the edges in the subtour can be used.

- Actually, we only need to consider subtours of size up to $n/2$ (where n is the number of cities), because if there is a larger subtour then the rest of the nodes must be contained in a smaller one (this might need some thought).

The travelling salesman problem (TSP)

- The drawback with this IP is that there are a lot of subtour elimination constraints: $2^{n-1} - n - 1$ of them. So this is not practical except for small TSPs. At least not with the usual solution methods.
- But there is in fact a way of solving large TSPs with this model: instead of putting all the subtour elimination constraints into a file before solving the IP, they are generated as needed during search.
- This is just one model; there are a few different alternative IPs for the TSP.
- Some of these are easier to solve than others.

An application of the TSP

- The TSP may be used to model other problems that at first seem unrelated.
- For example we might need to drill a number of holes in a sheet of metal as quickly as possible.
- This is a symmetric TSP: we must visit each location on the sheet, stopping at each, and minimise the total distance travelled.
- If you notice that your problem reduces to the TSP then you know how it should be solved, and you can use existing software.

TSP Application: Scheduling aircraft for take-off

- There must be a gap between each take-off, because of turbulence and vortices from the engines.
- The gap depends on the types of plane: a large plane can take off sooner than a small one, because it's less sensitive to air currents.
- But a large plane creates more disturbance than a small one for the following plane.
- We want a schedule that allows every aircraft to take off, while minimising the total time.
- Such a schedule is a list of all the planes with no repeats, which is just like a list of cities in a tour.

TSP Application: Scheduling aircraft for take-off

- Travelling from the current city A to one of the next ones B, C, D etc. is equivalent to allowing planes B, C, D etc to take off after plane A
- The distance from A to B is equivalent to the time that must be left after A before B can take off.
- This problem has natural asymmetry: a large plane might have to wait for 1 minute after a small one, but a small one might wait 3 minutes after a large one.
- So the "distance" from A to B may be very different from the "distance" from B to A.

TSP Application: Scheduling aircraft for take-off

- For example, suppose we have 3 types of plane: Jumbo, Airbus and Cessna in decreasing order of both size and engine power (there is an Airbus bigger than a Jumbo, but let's ignore this):

first→ second↓	Jumbo	Airbus	Cessna
Jumbo	3	2	1
Airbus	4	3	1
Cessna	8	7	2

- Now suppose we have 2 Jumbos, an Airbus and a Cessna to schedule. Let's call these planes J_1, J_2, A, C.

- The "distances" are:

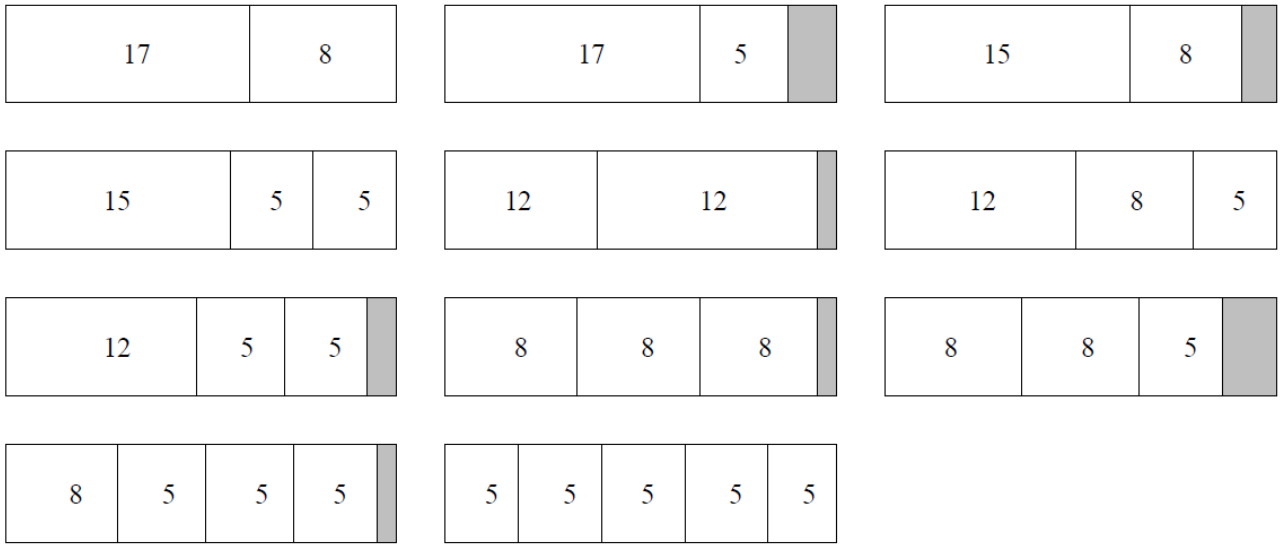
first→ second↓	J_1	J_2	A	C
J_1	—	3	2	1
J_2	3	—	2	1
A	4	4	—	1
C	8	8	7	—

TSP Application: Scheduling aircraft for take-off

- (We didn't use the Airbus-Airbus or Cessna-Cessna "distances" because there's only one plane of each type to schedule.)

Cutting stock problem

- A paper company sells rolls of paper of fixed length in 5 standard widths: 5, 8, 12, 15 and 17 feet.
- But the company can only make 25-foot-wide rolls, so all orders must be cut from rolls of this size.
- This can be done in 11 different ways:



(The patterns are numbered 1–11, row-by-row then column-by-column.)

Cutting stock problem

- Some of these patterns leave excess paper (shaded). I didn't include patterns whose excess was as great as 5 (the smallest standard width).
- The demands for the 5 widths are 40, 35, 30, 25 and 20 respectively.
- The problem is to cut the rolls into the required sizes to meet these demands while minimising the total number of rolls required.

Cutting stock problem

- We define 11 integer variables x_j : the number of rolls cut in pattern j. We want to minimise:

$$Z = x_1 + \ldots + x_{11}$$

- while satisfying the following 5 constraints, one for each demand for rolls of width 5, 8, 12, 15 and 17:

$$\begin{aligned} x_2 + 2x_4 + x_6 + 2x_7 + x_9 + 3x_{10} + 5x_{11} &\geq 40 \\ x_1 + x_3 + x_6 + 3x_8 + 2x_9 + x_{10} &\geq 35 \\ 2x_5 + x_6 + x_7 &\geq 30 \\ x_3 + x_4 &\geq 25 \\ x_1 + x_2 &\geq 20 \end{aligned}$$

Cutting stock problem

- Take the 3rd constraint, for example.
- The demand for rolls of width 12 is 30: each use of cutting pattern 5 generates two such rolls, and patterns 6 and 7 generated one each.
- No other cutting pattern generates this width, so this explains the left hand side $2x_5 + x_6 + x_7$.
- Here are 3 optimal solutions:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
19	1	9	16	14	2	0	1	0	2	0
16	4	7	18	15	0	0	4	0	0	0
20	0	5	20	15	0	0	4	0	0	0

Cutting stock problem

- Similar problems arise in timber and other industries.
- Actually, IP is not the best way to solve such problems, because the number of patterns is often huge.
- We would have to generate an impractically large set of constraints just to model the problem.
- There are ways of handling such situations (for example *column generation*) but we won't cover them in this course.

Problem 1

- A couple are planning a hiking trip together. The husband carries a knapsack able to hold 20 kg of equipment, while the wife carries another knapsack able to hold 17 kg. There are several items that they would like to take on the trip, each with a weight and an estimated "benefit" (given by a number):

Item	Weight	Benefit
stove	7	10
lamp	6	9
axe	13	6
binoculars	4	3
rope	9	14

Solve the problem using Pulp!

Problem 1 (cont.)

Once you have the solution to the problem above, you may add some extra constraints to ensure that:

- if the husband carries the stove then he also carries the lamp
- the wife carries the axe if and only if the husband carries the rope
- it is impossible to pack both the lamp and the rope, whichever knapsacks are used
- if the husband carries both the stove and the lamp then the wife carries the binoculars
- if the stove and lamp are both packed then so are the binoculars, whichever knapsacks are used