Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 6: Solving NP-hard problems via (Mixed) Integer Linear Programs

In this lab we will be exploring a general approach for solving NP-hard (and NP-complete) problems *exactly* in practice. This will be via the "magic box" of *(mixed) integer linear programming (MILP)*. An MILP is a program that has an *objective* function and a set of *constraints*. It tells us (or the computer) to optimise the value of the objective function making sure that none of the constraints is violated. This can be either a minimisation or a maximisation problem; in this lab we will be concerned with minimisation problems. The general form of a minimisation MILP is the following:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = 1, \ldots, m \\ & x_j \in \{0, 1\}, \qquad j = 1, \ldots, n \end{aligned}$$

Many NP-hard problems can be formulated as MILPs, by an appropriate choice of variables and constraints. If we manage to come up with such a formulation, we can use a *MILP solver* to find the optimal solution. You may think the MILP solver as a magic box that inputs the MILP formulation of your problem and returns an optimal solution. While that sounds fantastic, note that the magic box will not return an optimal solution in polynomial time - the problem that we formulated is NP-hard after all! What it does is that internally it runs a clever algorithm which takes exponential time in the worst case, but in practice runs faster than simply exhaustively searching through the possible solutions.

Remark: Before we proceed, let us remark the following. Suppose that instead of the general MILP form above, we had the following program:

$$\begin{aligned} \text{minimise} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = 1, \ldots, m \\ & 0 \leq x_j \leq 1, \qquad j = 1, \ldots, n \end{aligned}$$

This is called a general *linear program (LP)* formulation. Notice that the only difference between the MILP and the LP is that the former requires the variables to be either 0 or 1 (integers) whereas the latter allows them to be any real number between 0 and 1. Seems like a small difference, right? Well, this small difference makes all the difference in the world when it comes to solving these problems. Any problem that can be formulated as an LP is solvable in *polynomial time*. In fact, if you recall the 0/1-Knapsack of the lectures vs the fractional Knapsack of the tutorials, you may recall that the former is NP-complete whereas the latter can be solved in polynomial time via a greedy algorithm. The *integrality* of the variables which naturally arises from formulating NP-hard problems as optimisation programs makes them much harder to solve. Why exactly this is the case, you don't have to know right now.

Minimum Vertex Cover: In this lab sheet our goal will be to formulate the NP-complete (optimisation) problem Vertex Cover as a MILP, and write a Python program that inputs it to a solver and runs the solver to obtain a solution.

We will start with the formulation of the problem as a MILP. The important part is to identify the right variables x_j to use. Here, we will use *binary indicator* variables:

$$x_j = \begin{cases} 1, & \text{if } j \text{ is in the vertex cover} \\ 0, & \text{otherwise.} \end{cases}$$

From this, we can write the objective function as

$$\text{minimise} \sum_{j=1}^{n} x_j$$

Notice that if some node is in the vertex cover, then it contributes 1 to the sum, otherwise it contributes 0. So in the end the sum captures exactly the size of the vertex cover, and we are trying to minimise that.

Next we need to add the appropriate constraints. An obvious constraint is that $x_j \in \{0, 1\}$. Notice that if we do not use this, our solver could assign values e.g., $x_j = 0.6$, and then we cannot interpret this as a vertex cover: what does it mean for a node to be included in the vertex cover only by a 0.6-fraction? We want nodes to either be included or not. Finally, we need to add a constraint to make sure that each edge of the graph is covered, i.e., that at least one of its endpoints is in the vertex cover. Intuitively, we want something like

$$x_u = 1 \quad \texttt{or} \quad x_v = 1, \text{ for all } (u, v) \in E$$

This is something that our solver cannot handle however. The reason why MILPs are mixed integer ***linear*** programs is because all the constraints are linear functions of the variables, and so is the objective function. We cannot have an "or" function therefore, as it is not linear. Fortunatelly, we can write the constraint in a different way to achieve exactly the same thing:

$$x_u + x_v = 1, \text{ for all } (u, v) \in E$$

If at least one of x_u and x_v is set to 1, the constraint is satisfied and the edge is covered. If both of them are 0, then the constraint is violated. Our solver needs to return a feasible solution, so at least one of x_u and x_v will be set to 1 (possibly both).

In the end, we have the following MILP for Vertex Cover:

$$\begin{aligned} \text{minimise} \quad & \sum_{j \in V} x_j \\ \text{subject to} \quad & x_u + x_v \geq 1 \quad (u, v) \in E \\ & x_j \in \{0, 1\}, \quad j \in V \end{aligned}$$

Writing and solving MILPs in Python

As we said earlier, if we manage to formulate an NP-hard problem as a MILP, we can feed it to some MILP solver. There are many MILP solvers out there (e.g., CPLEX, Gurobi, etc), and each one might require the input to be provided in a slightly different format. So if you plan to use any of these solvers, you would need to read the documention to see how they expect the MILP to be represented. The most typical format is in terms of a constraint matrix A and vectors of coefficients c and b. This usually requires some effort, but they payoff is that it allows us to use the most powerful solvers out there.

For this lab sheet, we will work with an easier and more convenient representation, using the Python library `pulp` (see `https://coin-or.github.io/pulp/` for the documentation). To install pulp on your machine, write

```
python -m pip install pulp
```

On MacOS you might need to use instead:

```
python3 -m pip install pulp
```

A new variable (together with a constraint that it lies in $[0, 3]$) can be defined as follows:

```
x = LpVariable("x", 0, 3)
```

To define a binary variable, you may type:

```
x = LpVariable("x", cat=LpBinary)
```

Assume that we have a list `myList` of elements (e.g., integers) and we want to create a binary variable for each one of those. We can do that as follows:

```
x = LpVariable.dicts("x"myList,cat=LpBinary)
```

To define a minimisation problem to solve, we have:

```
prob = LpProblem("MyMinimisationProblem",LpMinimize)
```

Now let's assume that we want to add a constraint to our mininisation problem based on the variables that we have created. To add the constraint $x + y \leq 2$ we can simply write:

```
prob += x+y <=2
```

To solve the problem, we can write:

```
status = prob.solve()
```

To display the status of the solution, we can write `LpStatus[status]`.

We can access the value of a certain variable by `value(x)`. We can access the value of the objective function by `prob.objective.value()`.

See the documentation of the library for more details.

Exercise 1:
In this exercise, you are asked to write the Python code that formulates the MILP for VERTEX COVER in the format above (using the `pulp` library) for a given graph G. The graph is represented as a class with an Adjacency Matrix - you may use the code from the previous labs (e.g., in dijsktra.py) for this. In particular, your code should start with the following:

```
from pulp import *

class Graph():

def __init__(self, numNodes):
self.numNodes = numNodes
self.AdjacencyMatrix = [[0 for col in range(numNodes)] for row in range(numNodes)]
self.nodeSet = set(i for i in range(numNodes))

def __str__(self):
return str(self.AdjacencyMatrix)

def add_edge(self,node1, node2):
self.AdjacencyMatrix[node1][node2] = 1
self.AdjacencyMatrix[node2][node1] = 1

def delete_edge(self,node1,node2):
self.AdjacencyMatrix[node1][node2] = 0
self.AdjacencyMatrix[node2][node1] = 0
```

To write the MILP representation, do the following steps:

1. Define a minimisation problem called "VertexCover" using the `LpProblem` command.
2. Define binary variables x, one for each node of the graph, using the `x=LpVariable.dicts`(...) command.
3. Define the objective function as `prob += lpSum(x)`
4. For each node of the graph (i.e., for each pair of indices of the Adjacency Matrix for which the entry is 1), create the appropriate coverage constraint of the MILP and add it to `prob`.

5. Create a graph to test your solver on. You may for example use the following graph at first, but feel free to create more graphs to experiment.

```
inputGraph = Graph(8)
inputGraph.add_edge(0,1)
inputGraph.add_edge(0,3)
inputGraph.add_edge(0,5)
inputGraph.add_edge(1,2)
inputGraph.add_edge(2,4)
inputGraph.add_edge(2,7)
inputGraph.add_edge(3,6)
inputGraph.add_edge(4,7)
inputGraph.add_edge(5,6)
inputGraph.add_edge(6,7)
```

6. Solve the MILP using `prob.solve()`.

7. Define a set `indSet`, initialised to be empty. Iterate over all of the values of the variables in the solution of `prob.solve()`. For every value that is 1 (for numerical issues, you might want to consider every value that is larger than e.g., 0.99), add the corresponding node to `indSet`.

8. Output indSet by printing it on the screen.

Exercise 2:
In this exercise you will run your implementation above on a sequence of random graphs of increasing size. First you are asked to write a function

`randomGraph(numNodes,k)`

which creates a random graph with `numNodes` nodes, each of which has k neighbours. In particular, for each node u in the graph you will have to sample at random three distinct integers and add nodes between u and those nodes. You do not have to account for the fact that the random process might choose u to be a neighbour of u, as for large graphs this will happen with very small probability.

Once you have implemented this function, generate graphs with `numNodes` = 10, 50, 100, 500, 1000 and $k = 3$ and see how long the MILP solver takes to find an optimal solution. You should observe that as the sizes grow large, the program takes more time to finish. For some of the numbers above, you might want to terminate the execution if the solver does not finish in a long time (you can use `ctrl+C` for this).