graphs_and_lp

November 23, 2016

1 Graphs and linear programming

This tutorial will guide you through linear programming in Sage with an emphasis on graph theory. It is intended for beginners.

1.1 What is linear programming (LP)?

A linear program is the sum of two information:

- A linear function, called the objective, which is to be maximized or minimized (e.g. 2 x - y)
- Linear constraints on the variables (e.g. 3 x + y $\leq$ 2 and 5 x - 9 y = 1)

The solver will then try to find a solution to the system of constraints such that the objective function is optimized, and return the values of the variables.

Exercise: - Look at the documentation of `MixedIntegerLinearProgram` - Construct and solve the following linear program

```
* objective (to be maximized) 2 x - y
* constraint 1: 3 x + y &le; 2
* constraint 2: 2 x - 3 y &le; 8
* constraint 3: 8 x + y &ge; -8
```

- Draw the polyhedron associated to the constraints

```
In [ ]:
```

```
In [ ]:
```

1.2 What is a Mixed Integer Linear Program (MILP) ?

It is simply a Linear Program such that some variables are forced to take integer values instead of real values. This difference becomes very important when one learns that solving a Linear Program can be done in polynomial time while solving a general Mixed Integer Linear Program is usually NP-Complete (= it can take exponential time, according to a widely-spread belief that P is not equal to NP).

Exercise:

- Solve the same problem as in the previous exercise but using integer variables

- Plot the polyhedron of constraints together with the integer points in its interior
- Check that some of the corners are not lattice points and that the optimum is not reached on a vertex!

```
In [ ]:
```

```
In [ ]:
```

1.3 Why is Linear Programming so useful ?

Linear programming is very useful in many optimization and graph-theoretical problems because of its expressivity. Most of the time, a natural linear program can be easily written to solve a problem whose solution will be quickly computed thanks to the wealth of heuristics already contained in linear program solvers. It is often hard to theoretically find out the execution time of a Linear Program, though they give very interesting results in practice.

For more information, you can consult the Wikipedia page dedicated to linear programming : http://en.wikipedia.org/wiki/Linear_programming

1.4 Playing with graphs

Our aim is now to illustrate some usage of mixed integer linear program for graphs.

To build a graph (command `Graph`) or a digraph (command `DiGraph`) you just need to provide the adjacency as a dictionary. For example

```
G = Graph({0: [1, 2, 3], 1: [3], 2: [0,1]})
```

Once a graph is built you can access to many of its properties. Among them, we will mostly use

- `G.num_verts()`: for the number of vertices
- `G.num_edges()`: for the number of edges
- `G.vertices()`: the list of vertices
- `G.edges()`: the list of edges (together with labels)
- `G.neighbors(v)`: the list of the neighbors of a vertex `v`
- `G.plot()`: make a picture of the graph

Exercise:

- Build the graph proposed above and get various of its properties together with a picture (do not forget to access documentation and/or use tab completion)
- Build a directed graph and do the same

```
In [ ]:
```

```
In [ ]:
```

1.5 Prebuilt graphs

Sage also come with a huge list of prebuilt graphs that are available as `graphs.NameOfTheGraph()` as example

```
P = graphs.PetersenGraph()    # the Petersen graph
K = graphs.CompleteGraph(5)   # the complete graph on 5 vertices
```

and a bit of digraphs such as

```
D = digraphs.DeBruijn(2,3)    # the De Bruijn digraph on binary words of length 3
```

```
In [ ]:
```

```
In [ ]:
```

2 Vertex Cover in a graph

In the Vertex Cover problem, we are given a graph G and we want to find a subset S of its vertices of minimal cardinality such that each edge e is incident to at least one vertex of S.

In order to achieve it, we define a binary variable b_v for each vertex v. And minimize

$$\sum_{v\in G} b_v$$

subject to the constraints - $\forall v \in V(G)$, the variable b_v is binary - $\forall (u,v) \in E(G),\ b_u + b_v \geq 1$

Exercise:

- Check that the MILP formulation is actually correct
- Implement it in Sage and check it on some graphs

```
In [ ]:
```

```
In [ ]:
```

2.1 Maximum matching in a Graph

In the maximum matching problem, we are given a graph G, and we are looking for a set of edges M of maximum cardinality such that no two edges from M are adjacent.

We are considering a variable b_e for each edge e of the graph and maximize

$$\sum_{e\in E(G)} b_e$$

subject to the constraints - $\forall e \in E(G)$, the variable b_e is binary - $\forall v \in V(G)$, we have $\displaystyle\sum_{(u,v)\in E(G)} b_{uv} \leq 1$

Exercise: - Check that the MILP formulation is actually correct - Implement the maximum matching in Sage and check it on some graphs

```
In [ ]:
```

```
In [ ]:
```

2.2 Maximum Independent Set

A maximum independent set in a graph is a maximum set of vertices which are not connected to each other.

Exercise: - Find a MILP formulation for the maximum independent set problem - Implement this in Sage and check it on some graphs

```
In [ ]:
```

```
In [ ]:
```

2.3 Hamiltoninan path

We now explain how MILP can be used to solve a delicate problem: testing whether a graph is Hamiltonian. If you have a doubt about the definition, have a look at wikipedia. This program is a little more subtle than the previous one since conditions are dynamically added to the solver.

Let G be a graph in which we want to find a Hamiltonian path (or disprove that such path exists). We consider a binary variable b_e for each edge. As in the maximum matching we consider $H(b) = \{e \in E(G) : \ b_e = 1\}$ as a subgraph of G.

Exercise: - Formulate in terms of linear inequalities the fact that $H(b) = \{e \in E(G) : \ b_e = 1\}$ should be a disjoint union of cycles. - Find a linear equality that forces $H(b)$ to pass through all vertices. - Could you find linear inequalities that express that $H(b)$ is connected?

As you might have guess, connectivity is too many equations (with a lot of redundancy if not done carefully). The idea is to do not set these inequalities at the begining! We will just consider the MILP corresponding to a disjoint union of path H that pass through all vertices. Then we will repeat the following loop 1. ask for a solution $H(b)$ of the current problem 2. if $H(b)$ is connected we are done 3. otherwise, $H(b)$ determines a cut and we add a linear constraint that forbids this particular cut

Challenge: Write the above program!

```
In [ ]:
```

```
In [ ]:
```

This worksheet has been compiled from this very nice webpage of Nathann Cohen who sadly left Sage development. Some of the exercises actually correspond to how these problems are implemented in Sage. You can have a look at the source code of graphs.