Mathematics for Decision Making: An Introduction

Lecture 6

Matthias Köppe

UC Davis, Mathematics

January 22, 2009

Case Study: Line Drawings on Pen Plotters

Optimizing the operation of a pen plotter

Pen plotters are used instead of printers for very large-scale line drawings, such as for drawings in architecture, or charts of logic circuits in electronics. (Nowadays pen plotters are gradually being replaced by large-format inkjet printers.)

- The plotter can move a pen horizontally
- At the same time it can roll the paper (either a large sheet or paper from a roll) up and down
- These movements can be done in pen-up (not drawing) or pen-down (drawing) mode

Problem: Given a drawing to be produced, minimize the total drawing time.

Key insights:

- How is the drawing time determined?
- There are two parts of the total drawing time – one part is independent of our decisions, one does depend on our decisions.
- Can we draw every drawing in pen-down mode only?
- What are useful variables for modeling?
- What constraints do we need?

Pen plotter – Input data

A line drawing can be formalized like this:

- We assign numbers $1, \ldots, n$ arbitrarily to all endpoints of straight line segments and collect them in a set $V = \{1, \ldots, n\}$
- A line segment between endpoints i and j will be encoded by an edge $\{i, j\}$
- Thus the **combinatorics** of the drawing will be represented as a graph $G = (V, E)$
- In addition, we need as input the coordinates of the endpoints (they determine the objective function!); we denote them as parameters x_i, y_i for $i = 1, \ldots, n$.

Pen plotter: Assignment model

Finding an optimal way to draw a line drawing means to **bring actions into sequence**. Our first model therefore assigns **actions**, such as

- "draw a line from a to b"
- "move the pen (in pen-up mode) from a to b"

to **sequence numbers** (abstractions of passing time) $1, 2, 3, \ldots$.
Let's use 0/1 variables

$$x_{a,b,i} = 1 \text{ if we draw a line from } a \text{ to } b \text{ as the } i\text{-th action}$$
$$y_{a,b,i} = 1 \text{ if we move the pen from } a \text{ to } b \text{ as the } i\text{-th action}$$

It's clear we need at most $2n-1$ of these actions to draw n lines, so we use $i = 1, \ldots, 2n-1$.
Constraints: "We need to draw every line $\{a, b\}$ exactly once (direction does not matter)":

$$\sum_i (x_{a,b,i} + x_{b,a,i}) = 1 \qquad \text{for } \{a,b\} \in E$$

Constraints: "There is at most one action we can do at any given time":

$$\sum_{a,b \in V : a \neq b} y_{a,b,i} + \sum_{\{a,b\} \in E} x_{a,b,i} \leq 1 \qquad \text{for } i = 1, \ldots, 2n-1$$

Pen plotter: Assignment model, cont'd

Constraints: "No teleporting": We can start from point b in period $i+1$ only if we arrived at b in period i.

$$\sum_{\substack{a\in V:\\ a\neq b}} y_{a,b,i} + \sum_{\substack{a\in V:\\ \{a,b\}\in E}} x_{a,b,i} \geq \sum_{\substack{c\in V:\\ b\neq c}} y_{b,c,i+1} + \sum_{\substack{c\in V:\\ \{b,c\}\in E}} x_{b,c,i+1}$$

for $b \in V$ and $i = 1,\ldots,2n-2$.

By using $\geq$ (rather than $=$), we allow to let a period of activity be followed by a period in which nothing happens.

Notes on this model:

- The no-teleporting constraints can be understood as flow conservation constraints in a time-layered network with an implicit sink.
- The at-most-one-action constraint for period 1 together with the no-teleporting constraints imply all the other at-most-one-action constraints (flow principle).

More ZIMPL Power – Reading from data files

We want to be able to read parameters and sets from simple data files, to complete the separation between the ZIMPL model (which stays fixed) and data (which varies).

Following UNIX tradition, our data files are plain text files; one line is one “record” that we read in; an example:

Labeled points

```
# label x y
1 7 208
2 137 187
3 384 580
4 684 536
5 925 278
```

Edges in a graph

```
# from to
38 100
36 60
91 99
37 48
7 58
```

Here the line starting with # is a comment line. Non-comment lines are split into “fields”, separated by either whitespace or a comma, semicolon, or colon.

More ZIMPL Power – Reading from data files, II

ZIMPL uses this syntax, which can be used with `set` and `param` definitions:

```
    read FILENAME as TEMPLATE-STRING comment COMMENT-CHARACTER;
```

To read in the list of labeled points with coordinates, we first create a set of the labels.

```
set labels := { read "plot-points-100-1" as "<1n>" comment "#" };
```

Here the template string means “take the 1st field (of each line) and interpret it as a numeric (**n**) value”. The 1st field, in our data format, is the numeric label that we assign to the points.

Now that we have this index set, we read in the coordinates as (indexed) parameters. Note that parameters can only be numbers or strings (but not tuples), so we use separate parameters for the x and the y coordinates:

```
param x_coordinates[labels]
  := read "plot-points-100-1" as "<1n> 2n" comment "#";

param y_coordinates[labels]
  := read "plot-points-100-1" as "<1n> 3n" comment "#";
```

The template string means: Take the 1st field and interpret it as an index; then fill that parameter from the 2nd (or 3rd, respectively) field. Compare with the initialization syntax of parameters.

Modeling the Drawing Time (Objective Function)

Consider a straight line (to be drawn in pen-down mode) or a movement (in pen-up mode) from point (x_1, y_1) to point (x_2, y_2). Let $(\Delta x, \Delta y) = (x_1 - x_2, y_1 - y_2)$.

Pen-down movement

We assume that the drawing time for the line depends on $|\Delta x|$ and $|\Delta y|$, so let's denote it by $f_{\text{down}}(\Delta x, \Delta y)$.
Depending on the "pen" technology (ink pens vs. ballpoint pens or cutting knives), a plausible requirement could be that all lines are drawn with the same pen velocity α, to ensure the same amount of ink bleeds into the paper. So the drawing time would be proportional to the Euclidean distance of the endpoints:

$$f_{\text{down}}(\Delta x, \Delta y) = \alpha\sqrt{(\Delta x)^2 + (\Delta y)^2}.$$

Pen-up movement

We denote the movement time by $f_{\text{up}}(\Delta x, \Delta y)$.
In pen-up mode, horizontal movement (moving the pen) and vertical movement (rolling the paper up or down) can be done independently, at the maximum speeds β, γ of the two (different!) technologies.
A plausible model of the movement time therefore is:

$$f_{\text{up}}(\Delta x, \Delta y) = \max\{\beta|\Delta x|, \gamma|\Delta y|\}$$

Assignment model: Testing the formulation

Model and data:

- The complete ZIMPL model is found in `plotter-assignment.zpl`
- We use the data files named like this: `plot-points-10-1` and `plot-edges-10-20-1` – these are 10 points, and using a graph density of 20% (i.e., 20% of the edges of the complete graph are there)
- These data files were randomly generated using Python programs `make-random-points.py`, `make-random-graph.py`
- **Remark:** Generators of random examples allow to quickly test the performance of the model for various sizes of problems. . . but note that this might not tell enough about the performance of real-world examples, which could have completely different characteristics than randomly generated examples.

Assignment model – Test results and summary

Test results:

- The model (with 10 points and 10% density of edges to be drawn) creates about 1 800 variables, is solved in 6 seconds.
- 10 points, 20% density: about 2 000 variables, 79 seconds
- 10 points, 50% density: about 2 300 variables; after 680 seconds the best known feasible solution has 13.5% gap
- 30 points, 10% density: about 16 000 variables
- 50 points, 10% density: about 260 000 variables
- Models with 100 points or larger already use up a lot of virtual memory while interpreting the ZIMPL file

Summary:

- This model does not perform sufficiently well.
- (This is quite typical for **scheduling problems** modeled like this.)
- In the model above, we have very many variables because we wanted to encode a complete drawing algorithm (telling which line to draw, which movement to make, at each period).
- In the next step, we will try to reduce the number of variables.

Reducing the model – Decomposition

An important observation is that we can actually make the optimal decision for the drawing algorithm in **two steps**:

1. We first ignore the precise assignment of actions to periods, and just compute **which actions** should be taken. – So we now use integer variables

   $$x_{a,b} = 1 \text{ if we draw a line from } a \text{ to } b$$
   $$y_{a,b} = \text{number of times we move the pen from } a \text{ to } b$$

   (note $y_{a,b}$ is not a 0/1 variable).
2. Next we decide **when to schedule** these actions.
   All feasible ways to schedule the actions have the same cost (drawing time), so we just need to find one possible schedule.
   This is known as the **Euler walk** problem.
   We can do this either
   - with a simple combinatorial algorithm (homework)
   - with another integer programming model that is just like the old assignment model, but only has variables $x_{a,b,i}$ and $y_{a,b,i}$ for the actions actually taken (much fewer variables!)

This "decomposition" is possible because the cost (contribution to drawing time) of an action is independent of *when* the action is done.

The flow formulation for Step 1

Constraints: “We draw every line $\{a,b\}$ exactly once (direction does not matter)”:

$$x_{a,b} + x_{b,a} = 1 \qquad \text{for } \{a,b\} \in E$$

Constraints: “If we enter a vertex, we must leave it again (and conversely).” (These constraints are the remains of the no-teleporting constraints of the larger model.)

We need to make special exceptions for beginning and end – but how do we know at which point to begin and at which to end?

The solution is to add a beginning (“artificial source”) s and an end (“artificial sink”) t:

$$\begin{aligned}\bar{V} &= V \cup \{s,t\}\\ \bar{A} &= \{\,(a,b) \in V \times V : a \neq b\} \cup \{\text{arcs from } s\} \cup \{\text{arcs to } t\}\end{aligned}$$

The constraints then read:

$$\sum_{\substack{a\in\bar{V}:\\ (a,b)\in\bar{A}}} y_{a,b} + \sum_{\substack{a\in V:\\ \{a,b\}\in E}} x_{a,b} = \sum_{\substack{c\in\bar{V}:\\ (b,c)\in\bar{A}}} y_{b,c} + \sum_{\substack{c\in V:\\ \{b,c\}\in E}} x_{b,c} \quad \text{for } b \in V.$$

Of course, we need to make sure (in the objective function) it costs nothing to move from the artificial source to any point, and from any point to the artificial sink.

The flow formulation for Step 1

Constraint: Finally, we need to make sure that we move from the artificial source to exactly one real beginning – otherwise, we could “cheat” and start at many points “at the same time”!

$$\sum_{b \in V:(s,b) \in \bar{A}} y_{a,b} = 1.$$

The model is found in `plotter-flow.zpl`

- 10 points, 10% density: 119 variables, 0.02 seconds
- 10 points, 20% density: 127 variables, 0.05 seconds
- 10 points, 50% density: 145 variables, 0.07 seconds
- 30 points, 10% density: 1009 variables, 0.29 seconds
- 50 points, 10% density: 2771 variables, 16.7 seconds
- 50 points, 50% density: 3417 variables: after 2.7 seconds solution at most 0.03% away from the optimum, after 1000 seconds still a gap of 0.02% (and going)

General structure: Flows in networks

General structure: Flows in networks

- In general, consider a **digraph** (directed graph) (V, A), where
  - V is a set of vertices,
  - A is a set of (directed arcs), which are (ordered) pairs $(a, b) \in V \times V$ with $a \neq b$ (no loops!)
- We call two designated vertices $r, s \in V$ the **source** and the **sink**
- An r-s **flow $\mathbf{x}$** is a vector of real values $x_{a,b}$ for every arc $(a, b) \in A$ such that, in every vertex (except for source r and sink s), **flow conservation constraints** are satisfied:

$$f_{\mathbf{x}}(b) := \sum_{\substack{a \in V: \\ (a,b) \in A}} x_{a,b} - \sum_{\substack{c \in V: \\ (b,c) \in A}} x_{b,c} = 0 \quad \text{for } b \in V,\ b \neq r, s.$$

- We call $f_{\mathbf{x}}(b)$ the **excess** of the flow **x** at vertex b.
- The excess $f_{\mathbf{x}}(s)$ at the sink is called the **value** of the flow **x**; note $f_{\mathbf{x}}(s) = -f_{\mathbf{x}}(r)$.
- Often, **capacities** $u_{(a,b)}$ are given, i.e., upper bounds on the flow values $x_{(a,b)}$. We call a flow **x feasible** if it is non-negative and respects the upper bounds (if given).

General structure: Flows in networks

Maximum flow problem

Given a digraph (V, A), source r, sink s, arc capacities $u_{a,b}$:
Find a feasible flow $\mathbf{x}$ of maximum value $f_{\mathbf{x}}(s)$.

Minimum-cost r-s flow problem

Given a digraph (V, A), source r, sink s, arc capacities $u_{a,b}$, per-unit costs $c_{a,b}$, and a flow value ϕ:
Find a feasible flow $\mathbf{x}$ of value $f_{\mathbf{x}}(s) = \phi$ that has minimum total flow costs $\sum c_{a,b} x_{a,b}$.

- Variants of this problem with real flows and integer flows are both useful
- Flows appear naturally in many applications, such as:
    - Transportation problems through road networks
    - Water networks, oil pipelines
    - Telecommunication networks
- Flow formulations have good mathematical properties (total unimodularity), which make them an excellent modeling tool even for problems that do not look like flow problems
- The **shortest path problem** has a natural formulation as a minimum-cost flow problem, where we send one unit of flow from source to sink

Homework

1. Describe an algorithm that, given a graph (V, E), decides whether it has an Euler walk and, if so, constructs such a walk. Explain why your algorithm is correct.
2. Modify and test the TSP model.
   - Modify the TSP model (`tsp6-5.zpl`), so that it reads cities and their Euclidean coordinates from a data file.
   - Then compute the Euclidean distances of the cities within ZIMPL, rather than using the hard-coded distances of the 6-city TSP.
   - Create coordinate files for TSP problems of different sizes (at least 3 sizes, between 5 and 20 cities); for each size, create at least 3 different coordinate files
   - Test the performance of SCIP on this model, and find out how the computation time depends on the size, and how much it varies between different coordinate files for one size