Column Generation for Better Scalability

Scalability Issues

The main drawback of the cycle formulation is the limited scalability

- The number of cycles grows with the graph size as $O(n^{\text{max length}})$
- The enumeration becomes more expensive and the model becomes larger

Both can quickly become major bottlenecks

In [3]:
```
pairs2, arcs2, aplus2 = util.generate_compatibility_graph(size=150, seed=2)
print('>>> Size 150, enumeration time')
%time cycles2 = util.find_all_cycles(aplus2, max_length=4, cap=None)
print(f'Number of cycles: {len(cycles2)}')
print('>>> Size 150, solution time')
%time _, _, _ = util.cycle_formulation(pairs2, cycles2, tlim=10, verbose=0)
```

```
>>> Size 150, enumeration time
CPU times: user 7.29 s, sys: 7.12 ms, total: 7.3 s
Wall time: 7.31 s
Number of cycles: 43206
>>> Size 150, solution time
CPU times: user 1.64 s, sys: 43.5 ms, total: 1.68 s
Wall time: 1.68 s
```

Essentially, we have too many variables?

How can we address this?

Incremental Addition of Variables

We'll see how we can introduce variables as needed

Let us assume we have an optimization problem over a non-negative variable:

$$\operatorname*{argmin}_{x \geq 0} f(x)$$

- We assume that $f(x)$ is convex, which would make the problem easy
- ...Except that x is so large-dimensional that we cannot scale

Incremental Addition of Variables

We'll see how we can introduce variables as needed

Let us assume we have an optimization problem over a non-negative variable:

$$\operatorname*{argmin}_{x \geq 0} f(x)$$

- We assume that $f(x)$ is convex, which would make the problem easy
- ...Except that x is so large-dimensional that we cannot scale

We could obtain a solution quickly as follows:

- We restrict to a small subset S of the components of x, i.e. x_S
- ...We fix to 0 all other variables, i.e. $x_j = 0$ if $j \notin S$
- ...Then we find an optimum x_S^* via any suitable approach

But how do we know whether x_S^* is optimal?

Pricing

Since f is convex and there is no constraint optimality holds iff:

$$\nabla_x f(x^*) \geq 0$$

We check $\geq$, rather than $=$, since the variable is non-negative

- Since we have optimized over x_S, we know that $\nabla_{x_S} f(x_S^*) \geq 0$
- ...But what about the components that we kept out?

Pricing

Since f is convex and there is no constraint optimality holds iff:

$$\nabla_x f(x^*) \geq 0$$

We check $\geq$, rather than $=$, since the variable is non-negative

- Since we have optimized over x_S, we know that $\nabla_{x_S} f(x_S^*) \geq 0$
- ...But what about the components that we kept out?

We can still check their partial derivative after the problem is solved!

- If $\partial f(x^*)/\partial x_j \geq 0$, then we are also optimal w.r.t. x_j
- Otherwise, adding j to S might improve the solution

This post-solution derivative check is sometimes called pricing

Pricing for Incremental Variable Addition

So, we a criterion to find which variables should be added

In principle, we could proceed as follows:

- Choose S and we solve over x_S
- Loop over all $j \notin S$ and check $\partial f(x^*)/\partial x_j$
- Add the non-optimal variables to S and repeat until $\nabla_x f(x^*) \geq 0$

Pricing for Incremental Variable Addition

So, we a criterion to find which variables should be added

In principle, we could proceed as follows:

- Choose S and we solve over x_S
- Loop over all $j \notin S$ and check $\partial f(x^*)/\partial x_j$
- Add the non-optimal variables to S and repeat until $\nabla_x f(x^*) \geq 0$

This is a nice method, but it has one potential weakness:

- If we need to enumerate to check $\partial f(x^*)/\partial x_j$
- ...That may still take way too much time

We need a way to do the derivative check more efficiently

From Variable Addition to Variable Generation

Let's focus on decision variables representing complex entities

...Which can be constructed based on simpler building blocks

- E.g. cycles including several nodes
- E.g. routes including several arcs

From Variable Addition to Variable Generation

Let's focus on decision variables representing complex entities

...Which can be constructed based on simpler building blocks

- E.g. cycles including several nodes
- E.g. routes including several arcs

We can formalize this situation as follows:

$$x_j = g(y_j)$$

Given a variable y_j that specifies which building blocks are used:

- E.g. which nodes are included in the j-th cycle
- E.g. which arcs are included in the j-th route

...The function $g(y)$ specifies how a x_j is built

Pricing Problem

In these cases, we can avoid enumeration by using optimization

- First, we compute in closed form the derivative $\partial f(x^*_S)/\partial g(y)$
- Then we solve the pricing problem:

$$y^* = \operatorname*{argmin}_y \frac{\partial f(x^*_S)}{\partial g(y)}$$

Pricing Problem

In these cases, we can avoid enumeration by using optimization

- First, we compute in closed form the derivative $\partial f(x_S^*)/\partial g(y)$
- Then we solve the pricing problem:

$$y^* = \operatorname*{argmin}_y \frac{\partial f(x_S^*)}{\partial g(y)}$$

The result is the "recipe" y^* for the best possible variable $g(y^*)$

- I.e. the set of nodes leading to the best cycle
- I.e. the set of arcs leading to the best route

...And we can check the corresponding partial derivative as before

Column Generation

Let's revisit the process with this change

- Choose S and solve a restricted master problem over x_S
- Solve the pricing problem:

$$y^* = \operatorname*{argmin}_y \frac{\partial f(x_S^*)}{\partial g(y)}$$

- If $\partial f(x_S^*)/g(y^*) \geq 0$: the solution is optimal
- Oherwise: add $g(y^*)$ to the set variables and repeat

This process is called variable generation, or more often column generation

...Because in Linear Programming variables are associated to columns

The cycle formulation has a lot of variables...

Can we use this method for our use case?

CG for the KEP

We can try to use it for the relaxed cycle formulation:

$$\min -\sum_{j=1}^{n} w_j x_j$$

$$\text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq 1 \qquad \forall i = 1..m$$

$$x_j \geq 0 \qquad \forall j = 1..n$$

- All integrality constraints are relaxed, making the problem convex
- Note that $x_j \leq 1$ is implied by the other problem constraints
- We also changed the optimization direction to match the CG theory

CG for the KEP

Now we need to specify several things

- How to solve the restricted master problem
- How to compute the partial derivative $\partial f(x^*_S)/\partial x_j$
- Which basic decisions y to use for constructing a solution
- How those decision affect the variable buing built, i.e. the $g(y)$ function
- Finally, we need to define the pricing problem

We'll tackle one step at a time

CG for the KEP: Restricted Master Problem

Solving this problem for a subset S of variable is easy:

$$\min -\sum_{j\in S} w_j x_j$$

$$\text{s.t.} \sum_{j\in S} a_{ij} x_j \leq 1 \qquad \forall i = 1..m$$

$$x_j \geq 0 \qquad \forall j = 1..n$$

- Rather than setting all other variables to 0
- We just restrict the summations

This is much cheaper in terms of memory usage and solution time

CG for the KEP: Partial Derivative

The tricky part is computing $\partial f(x^*_S)/\partial x_j$

At a first glance, this seems easy:

- By differentiating:

$$-\sum_{j=1}^{n} w_j x_j$$

- We simply get:

$$-w_j$$

- However, unlike in our theoretical formulation
- ...Our problem has additional constraints

CG for the KEP: Partial Derivative

In particular, we have node mutual exclusion

$$\min -\sum_{j=1}^{n} w_j x_j$$

$$\text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq 1 \qquad \forall i = 1..m$$

$$x_j \geq 0 \qquad \forall j = 1..n$$

- In the optimal solution, some gradient component can be > 0
- ...Because the constraint prevents from moving in that diretion

How can we account for this?

CG for the KEP: Partial Derivative

Linear Programs satisfy strong duality

This means that the constraints can be turned into cost terms:

$$\min \mathcal{L}(x, \lambda) - \sum_{j=1}^{n} w_j x_j + \sum_{i=1}^{m} \lambda_i \left(\sum_{j \in S}^{n} a_{ij} x_j - 1 \right)$$

$$\text{s.t. } x_j \geq 0 \qquad \forall j = 1..n$$

The new cost function $\mathcal{L}(x, \lambda)$ is called a Lagrangian

- It is possible to define Lagrangian (or dual) multipliers λ_i
- S.t. $\nabla \mathcal{L}$ behaves like a normal gradient for an optimal solution

In fact, all LP solvers are capable to returning those λ

CG for the KEP: Partial Derivative

So, we differentiate $\mathcal{L}$ rather than the original cost

$$\frac{\partial \mathcal{L}(x_S^*)}{\partial x_j} = -w_j + \sum_{i=1}^{m} \lambda_i^* a_{ij}$$

- Where λ_i^* are the optimal multipliers for the x_S^* solution
- Again, they are computed automatically by the solver

I.e. this derivative is for one specific solution!

This partial derivative is called a reduced cost

- Reduced costs can be computed by using standard formulas
- ...But here we have derived them step by step

CG for the KEP: Building Variables

Now we need to specify how to build a variable, i. the $g(y)$ function

$$\min -\sum_{j=1}^{n} w_j x_j$$

$$\text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq 1 \qquad \forall i = 1..m$$

$$x_j \geq 0 \qquad \forall j = 1..n$$

The basic decision y_i consists in choosing whether to include node i

- When we set $y_i = 1$, for a previously unused variable x_j
- ...We increase w_j by 1 and we set $a_{ij} = 1$

CG for the KEP: Pricing Problem

The goal of the pricing problem is to minimize $\partial f(x_S^*)/\partial g(y)$

In practice this mean computeing the reduced cost:

$$\frac{\partial f(x_S^*)}{\partial x_j} = -w_j + \sum_{i=1}^{m} \lambda_i^* a_{ij}$$

...Expressed as a function of $\boldsymbol{y}$:

$$\frac{\partial f(x_S^*)}{\partial g(y)} = -\underbrace{\sum_{i=1}^{m} y_i}_{w_j} + \sum_{i=1}^{m} \lambda_i^* y_i$$

- This is true since we can decide which nodes to include

CG for the KEP: Pricing Problem

Overall, our pricing problem is as follows:

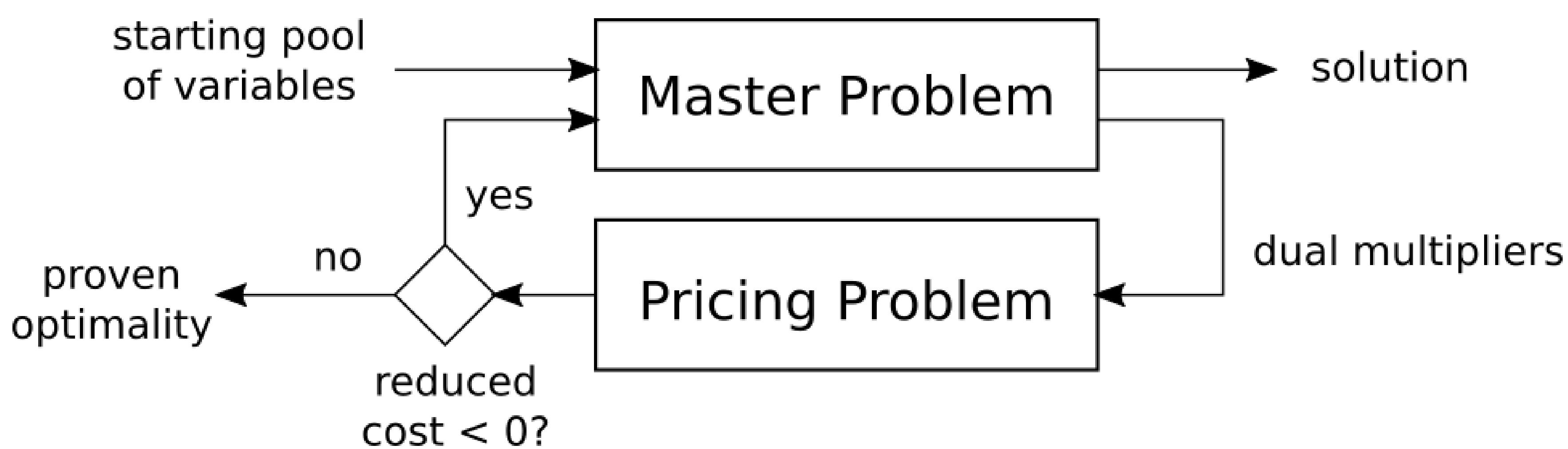
$$
\begin{aligned}
& \text{argmin} \sum_{i=1}^{m} y_i(-1 + \lambda_i^*) \\
& \text{s.t. } y \text{ defines a cycle} \\
& \sum_{i=1}^{m} y_i \leq C \\
& y_i \in \{0, 1\} \qquad\qquad \forall i = 1..m
\end{aligned}
$$

- Our selection nodes should have minimal weight
- ...It should define a cycle
- ...And it should not be too large

Column Generation

Overall, this is how CG is set up for Linear Programs

...Which are by far the most common application case:



- After every MP iteration, we obtain the dual multiplier
- ...Then we solve the pricing problem to obtain the best possible variable
- If the corresponging reduced cost is ≥ 0, we proved optimality
- Otherwise, we keep on looping