PRINT JOB OPTIMIZATION

JAMES JONES

Abstract. We try to formalize a print job optimization problem brought to the meetup iand come up with a reasonably quick way to, given a print job, determine how to print it at minimal cost.

1. Definitions and Presumptions[1]

1.1. Job and Version. A job j is a pair (V, q) where V is a non-empty, finite set of versions, identically-sized images of pieces to be printed, and $q : V \to \mathbb{Z}^+$ is a function indicating how many of each version the customer wants printed. We presume that our printer can print sheets of paper large enough to let us print at least one piece. For a given job, we define U to be the number of pieces that can fit on a sheet.

1.2. Form and Run. A form is a function $f : V \to \mathbb{N}$ such that $0 < u(f) \leq U$ where $u(f) = \sum_{v \in V} f(v)$, i.e. a form must print something and fit on a sheet. $f(v)$ is the number of pieces of version v printed per sheet for form f. A form f is said to be "$u(f)$-up". We call the set of possible forms for a job $\mathbb{F}$, and define $\pi : \mathbb{F} \to \mathscr{P}(\mathrm{V})$ by $\pi(f) = f^{-1}(\mathbb{Z}^+)$.

A form f corresponds to plates that print $\pi(f)$. The printer uses process color[4], so each form requires four or eight plates for single-sided or double-sided printing respectively.

A run is the process of preparing the plates corresponding to a form and printing a number of sheets using it.

1.3. Solution. A solution for a job (V, q) is a pair (F, p) such that $\emptyset \subset F \subseteq \mathbb{F}$ and $p : F \to \mathbb{Z}^+$ satisfies $\forall v \in V$, $q'(v) \geq q(v)$, where $q'(v) = \sum_{f \in F} p(f) f(v)$. p maps a form to the number of sheets to be printed with it, and q' maps a version to the total number of copies that will be printed with the specified solution, so the constraint just means "we print at least as many of each version as the customer wants". Given a solution (F, p), for each $f \in F$, one does a run printing $p(f)$ sheets using the plates created for f.

Clearly if (F, p) solves (V, q), $(F, v \mapsto kp(v))$ solves $(V, v \mapsto kq(v))$ for all $k \in \mathbb{Z}^+$. One can thus solve (V, q) by solving $(V, v \mapsto \frac{q(v)}{\gcd\, q[V]})$. If the q values tend to be round numbers, the smaller job may be easier to solve.[2]

2. The cost function

We don't just want solutions; we want solutions that minimize some cost function $c(F, p)$. What should the cost function reflect?

Date: May 30, 2018.

[1]We have revised this document for consistency, to the extent possible, with terms used in [1] and [2]. An appendix describes possibly-unfamiliar notation.

[2]That property helps one work out examples by hand, but the code won't need to bother. This is just as well, because an order often includes "seed" copies. These go to people who will confirm receipt, providing evidence that a mailing actually took place.

2.1. Wasted paper. Wasted paper comes in two flavors: blank piece-sized spaces on sheets and excess pieces printed. There are $\sum_{f\in F} p(f)(U-u(f))$ of the former and $\sum_{v\in V}\left(q'(v)-q(v)\right)$ of the latter, and their sum reflects the total wasted paper.

2.2. Wasted ink? The printer who brought us this problem said that ink wasn't a concern, so we'll ignore it; besides,

- Without detailed knowledge of what's being printed, we can't tell how much ink we're wasting anyway, but...
- ...wasted ink is ink printed on wasted paper, so minimizing wasted paper minimizes wasted ink. One could argue for weighting printed-on waste more heavily than blank waste, but that requires the detailed knowledge we lack.

2.3. Forms. Creating forms is, we're told, the resource-intensive part of an offset print job, and looking at videos of the process just for small sheets, I can believe it [6]. For double-sided printing (which the example we were first given requires), each form requires eight frames made from the CMYK separations for each side.

2.4. Weighting. The obvious cost function is, then, a weighted sum of wasted paper and the number of forms...but what are the weights, or equivalently, how many sheets would you waste to avoid another form?

Probably quite a few, from looking at how plates are prepared and the cost of plates and materials. One example of what, at least to me, seems a large job, involved printing nearly 800,000 pieces with 186 versions. It was printed 12-up and sheetwise, with thirty forms, at a cost per sheet of $0.20 and setup cost per press run of $475. At $0.20/sheet, that would pay for 2,360 sheets of paper, or $2360U$ pieces, so you'd have to seriously reduce waste to make it worth another form.

2.5. Why not just use the actual cost? Maybe we should. I was trying to reflect how far away a solution is from the optimum, but there's no way to express wasted forms without actually knowing the optimal solution. (It did show what an optimal one-form solution must be.)

3. Solutions

3.1. Maximum forms. Given a job (V,q), $(\{w\mapsto U\delta_{vw}|v\in V\},\ v\mapsto\left\lceil\frac{q(v)}{U}\right\rceil)$ is a solution with $\|V\|$ forms.[3] If $\|V\|$=1 or $U=1$, that is the solution. In general, it's pretty expensive, but at least it serves as an existence proof.

3.2. Minimum forms. Every job has a solution with $\left\lceil\frac{\|V\|}{U}\right\rceil$ forms. For simplicity's sake, let's start with the simple case, and then add the minor complication.

3.2.1. $U\mid\|V\|$. If U divides $\|V\|$, there are $\frac{\|V\|}{U}$ forms, all stuffed to the gills. More formally, $\forall f\in F, \|\pi(f)\|=U$, so $\forall v\in f,\ f(v)=1$ and hence $q'(v)=p(f)=\max q[\pi(f)]$. f thus adds $p(f)\sum_{v\in\pi(f)}\left(p(f)-q(v)\right)$ to the paper portion of the cost. Note that if $\|q\left[\pi(f)\right]\|=1$, said contribution is zero, which suggests printing versions with the same quantity together on a form.

3.2.2. $U\nmid\|V\|$. Otherwise, the forms are just as described above, plus one more with $\|\pi(f)\|<U$. This extra form and its leftover slots can be handled as described in 3.3.2.

[3]Here we use a flavor of the Kronecker delta[5] that compares versions rather than integers.

3.3. The one-form case. As long as $\|V\| < U$, we can print everything with one form. We'll discuss whether we should later.

3.3.1. The perfect solution. A perfect solution uses one form and prints exactly what the customer wants. If there's an integer p' such that $\forall v \in V$, $p' | q(v)$ and $\sum_{v \in V} \frac{q(v)}{p'} = U$, then $(\{f\},\ f \mapsto p')$, where $f(v) = \frac{q(v)}{p'}$, is the perfect solution, because

- $U = \sum_{v \in V} \frac{q(v)}{p'} = \sum_{v \in V} f(v) = u(f)$, so f is U-up; all printed sheets are full.
- $\forall v \in V$, $q'(v) = \sum_{f \in F} p(f) f(v) = p' \frac{q(v)}{p'} = q(v)$, so no excess copies are printed and no ink is wasted.
- One form is the best we can do.

$\sum_{v \in V} \frac{q(v)}{p'} = U \Rightarrow p' = \frac{\sum_{v \in V} q(v)}{U}$, so just calculate that value. If it's an integer that divides all the quantities, you're golden. If not, there will be some waste.

3.3.2. Pretty good solution. A perfect solution may not be possible, but one can come close. Consider the example from the first meetup where we talked about the problem, with

- $V = \{v_1, v_2, v_3\}$
- $q(v_1) = 1100$, $q(v_2) = 1000$, $q(v_3) = 300$
- $U = 12$

For doing this by hand, we can turn it into a search for a solution for (11, 10, 3). I came up with $f(v_1) = 5$, $f(v_2) = 5$, $f(v_3) = 2$, so that printing three sheets gives you 15 of v_1 and v_2 and six of v_3. Three's necessary to get eleven copies of v_1.

We could stop there and print 300 sheets to accommodate the order—but we shouldn't. A scaled-up perfect solution is necessarily perfect. A scaled-up less-than-perfect solution gives an even less perfect solution for a larger problem, but if all versions have excess copies, there may be room to improve. 220 sheets gives us 1100 of each of v_1 and v_2, and 440 of v_3, resulting in 100 excess of v_2 and 140 excess of v_3. 240 excess copies at 12-up is 20 sheets of paper, only four dollars more expensive than a perfect solution, if one even exists. I spent maybe five minutes coming up with that form, but our goal is to avoid such work, so...

...let's try our perfect formula anyway. $11 + 10 + 3 = 24$, a multiple of 12, giving $p' = 2$. 11 and 3 are odd, so they fail the first constraint—but look at what we got for our hand-generated form. Integer division of the vs by 2 gives us 5, 5, and 1 which add to 11, one less than U. Using that leftover for anything other than v_3 forces the full 300 sheets to get 300 of v_3, so 5, 5, 2 looks like (and in fact is) the best we can do, suggesting that the formula can guide us even if we can't achieve perfection.

We were fortunate with that first example, because our p' did meet one constraint if not both. Let's try one that meets neither.

- $V = \{v_1, v_2, v_3\}$
- $q(v_1) = 1000$, $q(v_2) = 800$, $q(v_3) = 300$
- $U = 12$

10 + 8 + 3 = 21, and $12 \nmid 21$. $p' = 1$ fails, because 10 + 8 + 3 > 12. $p' = \left\lceil \frac{\sum_{v \in V} q(v)}{U} \right\rceil$ works whether or not the sum is a multiple of U. Then we have 5, 4, 1 and two leftover slots to play with. As before, at least one must go to v_3, or we need 300 sheets to get the needed copies of v_3. 200 sheets of 5, 4, 2 gives us 1000 of v_1, 800 of v_2, and 400 of v_3, wasting 100 printed copies and 200 blank spaces, equivalent to 25 sheets or $5. Further improvement requires more leftovers than remain.

The key: for an imperfect solution, no one p' is best for all versions. You have to use $p' = \max\left\{\left\lceil \frac{q(v)}{f(v)} \right\rceil \mid v \in V\right\}$, which in turn tells us where to hand out leftovers.

3.4. The general case. At the top level, we hand the one-form solver at most U versions at a time (if we don't, you can easily have all f values zero. Poof: nontermination!) and accumulate the forms it returns until we've handled all the versions.

3.5. Code and testing. About the Python code for the imperfect case: we use the Python heapq package, which is almost what we want. Since it implements a "min heap" and we want a "max heap", we keep the negative of the p' value in the heap and negate it at the end.

Testing turned up some bugs:

- The first version, which did the imperfect p' calculation in integer arithmetic, led to a division by zero when a small U was combined with widely varying quantities. We solved this by staying in floating point (and working around Python raising an exception rather than returning appropriately-signed infinity when dividing by zero).
- Still wider variation in quantities led to a situation with some zero f values even after distributing the leftovers. We handle that now by not putting the corresponding versions in the form.

We've yet to find a case in which adding a form reduces waste enough to make up for the setup costs of the additional form, as opposed to adding a form because we have no choice. Until we find such a case, we won't add code to try it.

Things that remain to be done:

- Testing, testing, testing. Test data from real life print jobs and the results of human estimators to compare against would be invaluable to exercise the code and suggest improvements.
- Refactoring as needed.
- Considering how best to return the solution, or possibly adding code to display the forms visually.
- Changing the top-level interface, e.g. how versions are represented, not requiring ordering by quantity

4. Real Data

We now have some real data to work with from a job with sixty-three versions, with $\sum_{v \in V} q(v) > 40000$ and $U = 12$. We've learned two things:

- Handing oneForm() U versions with quantities in descending order pretty well guarantees that you will get the minimum-form solution, where $\forall v \in V,\ f(v) = 1$. For the particular job we have data for, it only used five forms, but wasted ten times as many printed copies as the hand-generated solution.
- The higher-level code needs to be smarter. Suppose we pass oneForm() [16004, 16004, 4010, 4002, 4002] for quantities and $U = 12$. It will give back f values of 4, 4, 2, 1, 1 and $p' = 4002$. That means 6 + 6 + 7998 + 0 + 0 = 8004 printed copies wasted. If we instead chose 4010 for p', f values of 4, 4, 1, 1, 1 would suffice, and would result in 36 + 36 + 0 + 8 + 8 = 88 printed copies wasted plus 4010 blank space that would let us include a version with $q = 4004$, giving total waste of 88 + 6 = 94 printed copies, which would fit on 8 sheets of paper, $1.60 at $0.20/sheet. That, not to mention getting another 4004 printed copies on the form, beats the heck out of $133.80 paper cost and needing another form. (Passing the quantities with 4004

```
from typing import List, Tuple, Dict
from math import ceil, inf
from heapq import heapify, heappop, heappushpop

def solve(q: List[int], u: int) -> Tuple[List[Dict[int, int]], List[int]]:
  """The general case."""
  # preconditions: q is monotonically nonincreasing; all(qv > 0 for qv in q); u > 0
  F = [] # type: List[Dict[int, int]]
  p = [] # type: List[int]
  base = 0
  while base < len(q):
      f, pf = oneForm(q[base : base + min(len(q), u)], u)
      F.append(dict(zip((base + i for i in range(len(f))), f)))
      p.append(pf)
      base += len(f)
  return F, p

def oneForm(q: List[int], u: int):
    """Given q and U, return (f, p)."""
    # precondition: len(q) <= u and all(qv > 0 for qv in q)
    def pVal(qv: int, fv: int) -> Tuple[List[int], int]:
        return inf if fv == 0 else ceil(qv / fv)

    while True:
        idealP = sum(q) / u
        p = int(ceil(idealP))
        f = [float(qv // p) for qv in q]
        if p == idealP and all(qv % p == 0 for qv in q):
            break # perfect; we're done

        uncommitted = [] # type: List[int]
        pvHeap = list(zip((-pVal(qv, fv) for qv, fv in zip(q, f)), range(len(q))))
        heapify(pvHeap)
        p, v = heappop(pvHeap)
        for _ in range(u - int(sum(f))):
            uncommitted.append(v)
            f[v] += 1
            pNew, v = heappushpop(pvHeap, (-pVal(q[v], f[v]), v))
            if pNew > p:
                uncommitted = []
                p = pNew
        # Discard uncommitted leftovers.
        for v in uncommitted:
            f[v] -= 1
        if p != -inf:
            p = int(-p)
            break # for now, we'll call it good if it's feasible
        # Discard zeroes in f and corresponding q
        # TODO: add code to test our belief that the zeroes, if present, are
        # all at the end of f. Monotonicity of q *should* imply that, but...
        q, f = zip(*[x for x in zip(q, f) if x[1] != 0])

    return ([int(fv) for fv in f], p)
```

included to oneForm() does return this better solution, but the higher level code must be smart enough to try these values.)

From this we know

- $p' = \max\left\{\left\lceil \frac{q(v)}{f(v)} \right\rceil | v \in V\right\}$ is not required; that's a minimum value, and as the above example shows, a larger p' may, in imperfect cases, be preferable—but oneForm() can't know that.

- It can be better to use leftovers to pull in another version—but if we always pass along U values to oneForm(), it will never notice that.

That means we need another approach.

5. Approximate Scaled Partitions

Given a positive integer n, a partition of n is a multiset of integers that collectively add up to n. (Defining it as a multiset gets the behavior we want; $\{2, 2, 1\}$ and $\{2, 1, 2\}$ are the same partition of 5.)

Throughout the examples here you've no doubt noticed that the f values for a form often partition U. The second constraint in the perfect case is that the numbers that we ultimately choose as the f values partition U.Rather than take in a set of quantities and see how close to a partition of U they come, organize the quantities to make it easy to find partitions of U. If U doesn't get too large, it's easy enough to generate the partitions. Then for each version v, build a map from n to a list of other versions with quantities approximately either n or $\frac{1}{n}$ times v's quantity (whichever makes the code easier). Once you find that partition, you can calculate p' as above. (Will there always be a partition? Yes; if nothing else, there's $\{U\}$, which corresponds to the maximum forms case.)

6. Context

This is a special case of gang-run printing[8]. In the general problem, versions needn't all have the same size. (Some of the commercial packages listed on the Wikipedia page don't even require that they be rectangular.)

Appendix A. Notation and Definitions

Predicate calculus. We use $\Rightarrow$ for implication, and $\forall$ and $\exists$ for the universal ("for all") and existential ("there exists") quantifiers respectively.

Sets. We use upper case italic letters for sets, with some exceptions:

- $\mathbb{N}$, the set of natural numbers $\{0, 1, 2, 3, 4, ...\}$
- $\mathbb{Z}$, the set of integers, and $\mathbb{Z}^+$, the set of positive integers
- $\mathbb{F}$, the set of possible forms for a given job

Given a set S, $\|S\|$ is S's cardinality (number of elements), and $\mathscr{P}(\mathrm{S}) = \{\mathrm{S}' | \mathrm{S}' \subseteq \mathrm{S}\}$ is the power set of S.

Multisets. A multiset is a pair (S, m) where S is a set and $m : S \to \mathbb{Z}^+$is the multiset's "multiplicity" function; for any $s \in S$, $m(s)$ is the number of times s appears in the multiset. That's not how multisets are notated, though. We'll use the notation that is most like plain sets, with the only difference being that for a multiset, $\{a, a, b\} \neq \{a, b\}$. In the multiset on the left, a's multiplicity is 2, but in the one on the right, it's 1.

Functions. $f : A \to B$ means f is a function with domain A and codomain (aka "range") B. If we're defining a set of functions, as when we define forms, we give constraints the functions must meet. If we're defining a particular function, we specify how to determine the element of the codomain corresponding to a given element of the domain.

If the domain and codomain are clear from context, we needn't explicitly state them. For example, in the definition of forms, $u : \mathbb{F} \to \mathbb{Z}^+$ because u maps forms to necessarily positive (given the constraints on forms) sums of natural numbers.

We use $\mapsto$ instead of lambda notation, e.g. $v \mapsto \left\lceil \frac{q(v)}{U} \right\rceil$ for the anonymous form in the maximum forms solution.

Given $f : X \to Y$, we define $f^{-1} : \mathscr{P}(\mathrm{Y}) \to \mathscr{P}(\mathrm{X})$ as $f^{-1}(S) = \{x | f(x) \in S\}$ and the potentially confusing $f : \mathscr{P}(\mathrm{X}) \to \mathscr{P}(\mathrm{Y})$ as $f[S] = \{f(s) | s \in S\}$. The latter maps subsets of X to their "images" under f. The brackets help distinguish the two; think of it as polymorphism, "map f", or the equivalent in your language of choice.

Some functions can be extended via folds to operate on sets of values as well as pairs of values, e.g. min, max, and gcd. In that case we will omit the parentheses, e.g. gcd $\{q(v) | v \in V\}$ or, thanks to the preceding paragraph, gcd $q[V]$.

Miscellanea.

- For $m, n \in \mathbb{Z}$, we write $m|n$ or $m \nmid n$ to indicate that m does (or doesn't, respectively) divide n.
- $\lceil x \rceil$is the "ceiling" of x, i.e. min $\{z \in \mathbb{Z} | \mathrm{z} \geq \mathrm{x}\}$
- $\lfloor x \rfloor$is the "floor" of x, i.e. max $\{z \in \mathbb{Z} | \mathrm{z} \leq \mathrm{x}\}$

References

[1] https://github.com/edwja/gang-problem/gang.pdf
[2] PrintWiki, http://printwiki.org/Front_Page
[4] PrintWiki, http://printwiki.org/Process_Color
[5] https://en.wikipedia.org/wiki/Kronecker_delta
[6] "Intro Platemaking", https://www.youtube.com/watch?v=7hNXFG1oi-Q&t=325s
[7] PrintWiki, http://printwiki.org/Sheetwise
[8] Wikipedia, https://en.wikipedia.org/wiki/Gang_run_printing