A Generic Algorithm for Minimum Chain Partitioning

Sharon A. Curtis
Department of Computer Science
University of Stirling, Stirling FK9 4LA, UK
s.curtis@cs.stir.ac.uk

Abstract This presents a generic algorithm to obtain a minimum sized partition of a partially ordered set into chains. The algorithm is illustrated with three examples, including a novel box stacking problem and solution.

1. Introduction

A finite partially ordered set may be partitioned into *chains* (linearly ordered subsets). We consider the partitioning of such a set into the minimum possible number of chains.

This general problem occurs in many areas of combinatorics: indeed it is so common that the popular software package *Mathematica* [10] contains a function *MinimumChainPartition* specifically for this purpose. In 1950, a decomposition theorem of Dilworth [5] showed that the minimum size of such a partitioning is equal to the maximum size of an *antichain* (a pairwise incomparable subset), but the proof is not constructive, and so does not actually find such a partitioning.

In 1962, Ford and Fulkerson [6] showed that the complexity of the minimum chain partition problem is equivalent to that of finding a maximum bipartite matching, and in 1973, Hopcroft and Karp [9] produced a well-known matching algorithm with complexity $O(mn^{1/2})$ for a graph with n vertices and m edges. This was further improved in 1991 for dense graphs, by an algorithm due to Alt, Blum, Mehlhorn and Paul [1] with complexity $O(n^{1.5}\sqrt{m/log\ n})$.

Given that a lower bound for minimum chain partitioning is $\Omega(n^2)$ (as every pair of elements may need to be compared), there is still room for improvement. In this paper, we address some cases of this problem by applying (reasonably general) restrictions to the partial order,

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35672-3_13

J. Gibbons et al. (eds.), *Generic Programming*

and obtain a generic algorithm of worst case $O(n^2)$ running time. The algorithm is illustrated by the solution of three varied minimum chain partitioning problems: "Fixed-Job Scheduling", "Smallest Upravel", and "Box Stacking".

2. The Generic Algorithm

We present a generic algorithm for minimum chain partitioning of a finite set of elements partially ordered by a relation P, and consider conditions for its correctness later.

The first step of the generic algorithm is to sort the initial set of elements with respect to a linear ordering Q, which is chosen so that the result of the sorting is a topological sort of the elements with respect to P. That is, for every distinct pair of elements x, y related by P so that xPy, x must appear to the left of y in the list sorted with respect to Q. (Note that the anti-symmetry property of P will ensure the impossibility of both xPy and yPx.) Formally, this relationship between P and Q may be stated as $Q \cap \neq \ \subseteq \neg P^o$, where $\neg$ and ^{o} refer to negation and converse respectively. Having sorted the elements with respect to Q, the resulting list then has a desirable property: any chain of the original set will be a subsequence of this list.

The second and final step of the algorithm is to repeatedly extract chains from this list. The first chain extracted is the longest possible when starting with the leftmost element and moving from left to right along the list, adding any eligible elements to the end of the accumulating chain. Further chains are repeatedly extracted from the remaining elements in the same way.

To make this precise, here is the whole algorithm functionally expressed (with some efficiency sacrificed for clarity):

```
minChains :: (a->a->Bool) -> (a->a->Bool) -> [a] -> [[a]]
minChains p q  = pickChains p . sort q

pickChains :: (a -> a -> Bool) -> [a] -> [[a]]
pickChains p []     = []
pickChains p (x:xs) = chain : pickChains p remainder
      where (chain,remainder) = pickChain p [x] xs

pickChain :: (a -> a -> Bool) -> [a] -> [a] -> ([a],[a])
pickChain p ch []     = (ch,[])
pickChain p ch (x:xs)
      | p c x         = pickChain p (ch++[x]) xs
      | otherwise     = (chain, x:remainder)
```

```
where c = last ch
      (chain,remainder) = pickChain p ch xs
```

The worst-case complexity of this generic algorithm is $O(n^2)$: the sorting can be done in $\Theta(n \; log \; n)$, however the second step may find it necessary to compare every pair of elements (for example, when $P = \{\}$). This quadratic algorithm is thus an improvement on the best-known algorithms for minimum chain partitioning, but unfortunately it is not generally correct (it is easy to construct a counterexample where the minimum number of chains is not achieved). However, it is possible to place reasonable conditions on the orderings P and Q under which the algorithm is correct, and we now present these conditions together with a proof of correctness.

2.1. Conditions for Correctness

First we note that it does not matter whether P is the usual reflexive variety of partial order, or a *strict* (anti-reflexive) one: for chain construction, elements never need to be compared to themselves. Thus we assume only that P is transitive and anti-symmetric.

We have already stated one condition on Q, namely that sorting with respect to Q produces a topological sort with respect to P. The one further condition required will arise naturally from the proof of the following lemma:

Lemma 1 *Suppose that P and Q are orderings on a finite set, such that P is transitive and anti-symmetric, Q is linear, and the following two conditions hold:*

$$Q \cap \neq \subseteq \neg P^o \tag{1}$$

$$P \,.\, Q \cap (\neg P \cap Q)^o \,.\, P \subseteq P \tag{2}$$

Then the generic algorithm given above produces a minimum size partitioning into chains.

Proof

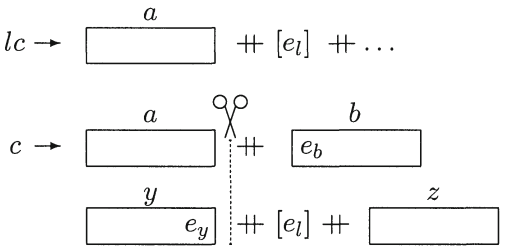
The method of proof is as follows: we consider a minimum chain partitioning, and transform it into another optimal chain partitioning containing the chain first selected by the algorithm. The argument can then be repeated for the remaining chain selections of the algorithm.

Suppose the original set of elements has been sorted with respect to Q to form the list x, and that the longest leftmost chain (as selected by `pickChain`) in x is lc. Suppose further that C is an optimal chain partitioning. As such, one of its chains, c say, must begin with *head* x (as also must lc).

We wish to change C into another chain partitioning containing lc, by performing changes that result in c being transformed into lc, without altering the optimality of the resulting partitioning. This is done as a result of the claim that if $c \neq lc$, it is possible to extend the longest common prefix of c and lc by one more element. Repeating this step completes the desired change.

Proof of claim:

Denote the longest common prefix of lc and c by a, so that $c = a +\!\!+ b$ for some b. As lc is a maximal chain and cannot be a prefix of c, lc must be of the form $a +\!\!+ [e_l] +\!\!+ \ldots$, and the element e_l must occur in some other chain $y +\!\!+ [e_l] +\!\!+ z$ of C. We extend the longest common prefix of lc and c by cutting the chains c and $y +\!\!+ [e_l] +\!\!+ z$, and rejoining them to form $a +\!\!+ [e_l] +\!\!+ z$ and $y +\!\!+ b$. This diagram illustrates the idea:



The chain property of lc, and that of $y +\!\!+ [e_l] +\!\!+ z$, ensures that $a +\!\!+ [e_l] +\!\!+ z$ is also a chain, so it remains to check that $y +\!\!+ b$ is a chain. If either $y = [\,]$ or $b = [\,]$, then $y +\!\!+ b$ is a chain. Otherwise, we let $e_b = head\ b$, and $e_y = last\ y$, and it is then sufficient to show that $e_y\ P\ e_b$.

Gathering information about e_y and e_b, we see that $e_y\ P\ e_l$ as $y +\!\!+ [e_l]$ is a chain, and also $e_l\ Q\ e_b$ (from the choice of e_l rather than e_b in the construction of lc) This gives us that $e_y\ (P.Q)\ e_b$.

We also consider where e_y appears in the original list x. The element e_y must occur to the left of e_l (as $y +\!\!+ [e_l]$ is a subsequence of x), but does not occur in $a +\!\!+ [e_l]$, so must appear between two adjacent elements of $a +\!\!+ [e_l]$, say between a_i and a_{i+1}. As the list is sorted with respect to Q, this ensures that $a_i\ Q\ e_y$; also as c is a chain, $a_i\ P\ e_b$. Furthermore, from the algorithm's choice of a_{i+1} rather than e_y, we have that $a_i\ \neg P\ e_y$. We have just shown that $e_y((\neg P \cap Q)^o\ .\ P)e_b$,

Combining these two pieces of information about e_y and e_b, we have that $e_y(P.Q \cap (\neg P \cap Q)^o\ .\ P)e_b$, and so from (2), $e_y\ P\ e_b$. □

Given a particular partitioning problem to solve, it is easy enough to think of a Q to topologically sort the elements with respect to P, but condition (2) looks more complicated, and is not sufficiently intuitive to

suggest which Q might be most suitable. The following corollary gives an easier alternative:

Corollary 2 *The following condition implies* (2)*:*

$$P \,.\, Q \;\subseteq\; P \tag{3}$$

This condition is much easier to verify than (2), but it is restrictive, and not all chain partitioning problems have an obvious suitable Q for which this is true. For other problems, we now take a closer look at the structure of P.

A common occurrence in chain partitioning problems is that P can be expressed as the intersection of two orderings. The following lemma gives simple conditions on those orderings, sufficient to satisfy (2):

Lemma 3 *Let P, Q, P_Q and P_R be orderings on a finite set, such that:*

$$P \;=\; P_Q \cap P_R \tag{4}$$
$$P_Q \,.\, Q \;\subseteq\; P_Q \tag{5}$$
$$\neg P_R^o \,.\, P_R \;\subseteq\; P_R \tag{6}$$
$$\neg P \cap Q \;\subseteq\; \neg P_R \tag{7}$$

Then condition (2) *holds.*

Proof

$$\begin{array}{ll} & P \,.\, Q \;\cap\; (\neg P \cap Q)^o \,.\, P \;\subseteq\; P \\ \Leftarrow & \quad \{(4), \text{intersection}\} \\ & P \,.\, Q \;\cap\; (\neg P \cap Q)^o \,.\, P \;\subseteq\; P_Q \\ & \wedge\; P \,.\, Q \;\cap\; (\neg P \cap Q)^o \,.\, P \;\subseteq\; P_R \\ \Leftarrow & \quad \{\text{intersection}\} \\ & P \,.\, Q \;\subseteq\; P_Q \\ & \wedge\; (\neg P \cap Q)^o \,.\, P \;\subseteq\; P_R \\ \Leftarrow & \quad \{(4), (7)\} \\ & P_Q \,.\, Q \;\subseteq\; P_Q \\ & \wedge\; \neg P_R^o \,.\, P \;\subseteq\; P_R \\ \Leftarrow & \quad \{(4), (5)\} \\ & \neg P_R^o \,.\, P_R \;\subseteq\; P_R \\ \Leftarrow & \quad \{(6)\} \\ & \mathit{true} \end{array}$$

□

In practice, P_Q is usually an ordering very similar to Q (hence the labelling of it as P_Q). This makes (5) very similar to a transitivity condition. Condition (6) is also similar to transitivity, but takes into account possible non-reflexivity. For example, if P_R is $<$, (6) would be equivalent to $\forall x, y, z \bullet x \not> y \ \wedge \ y < z \Rightarrow x < z$.

3. Examples

We now consider three varied examples of minimum chain partitioning problems, and their solution by the generic algorithm.

3.1. Fixed-Job Scheduling

Some scheduling problems can be expressed as partitioning problems involving time constraints. In particular, the well-known fixed-job scheduling problem (FSP) considers the allocation of a fixed number of jobs to resources. The jobs have fixed start and end times, and each resource can only cope with one job at once. The resources all have identical capabilities, and overall it is required to use as few resources as possible to complete all the jobs. The FSP is thus a minimum chain partitioning problem, where a chain contains a schedule of jobs for one of the resources.

The FSP has occurred frequently in the literature, in plain unadorned form [8], and for more practical situations. For example, aircraft maintenance might need to be scheduled for as few engineers as possible, where the arrival/departure times of the aircraft are known; Dantzig and Fulkerson [4] minimized the number of US Naval tankers used for a delivery schedule; Gupta, Lee and Leung [7] called it the "Optimal Channel Assignment Problem" and used it for LSI circuit board design; the FSP may also be used for neat and economical layout of a Gantt chart (a management tool which lays out a schedule of tasks, invented by Henry Gantt in 1917).

Formalizing the FSP, the required ordering is thus:

$$\forall\, j_1 j_2 \bullet \quad j_1 \; P \; j_2 \quad \equiv \quad end\; j_1 \leqslant start\; j_2,$$

which is indeed transitive and anti-symmetric. Next we must choose a linear ordering Q. It seems intuitively reasonable to choose to sort the available jobs based on start time, so that

$$\forall\, j_1 j_2 \bullet \quad j_1 \; Q \; j_2 \quad \equiv \quad start\; j_1 \leqslant start\; j_2,$$

and this satisfies condition (1). Checking the simpler condition (3) from Corollary 2, we find that it is trivially true, as $end\ j_1 \leqslant start\ j_3$ is implied by $end\ j_1 \leqslant start\ j_2 \leqslant start\ j_3$. Thus the generic algorithm

solves this problem, by first sorting the jobs with respect to their start time, and then repeatedly extracting leftmost longest chains from the resulting list.

It is interesting to compare this solution to that of Gupta, Lee and Leung in [7]. The first stage of their algorithm involves sorting not only the start times, but also the end times. The second stage of their algorithm does a plane sweep in linear time across these start/end times, using a stack to efficiently allocate jobs to resources. Altogether, the algorithm is of $O(n \ log \ n)$ complexity.

The generic algorithm is of $O(n^2)$ in the worst case, however it should be noted that this can be improved for the FSP. Firstly, this is achieved by re-expressing the second step of the algorithm so that the elements are dealt with in one pass from left to right (that is, expressing this step as a fold, rather than an unfold; see, for instance, [3] for details of these classifications). Secondly, the algorithm is changed to use an efficient data structure to hold the chains during their construction, sorted by the end time of the last job on each chain. This leads to an algorithm also of $O(n \ log \ n)$ complexity. Furthermore, it has the advantage of being an on-line algorithm, which is useful if the jobs to be scheduled are not known in advance, but are provided in real time. The plane-sweep algorithm of Gupta, Lee and Leung does not share this advantage.

3.2. Smallest Upravel

In [2], Bird defines a *ravel* of a sequence to be a partitioning into subsequences. The ravel is further an *upravel* if all the component subsequences are in ascending order. The problem is to find the smallest possible upravel of the original sequence. For example, the string "chocolate" has an upravel of {"choo", "ct", "l", "ae"}, but its smallest upravel is {"choot", "cl", "ae"}.

The difference between this problem and the original minimum chains partitioning problem is that the elements are given as a sequence, not a set. This may easily be overcome by labelling each element with its index position in the sequence, to obtain a set of unique elements, and so the ordering P is defined to be

$$P \ = \ \leqslant_{char} \ \cap \ \leqslant_{index}$$

Note that the elements of the sequence need not be characters: any totally ordered datatype for the elements will do.

As for Q, the elements must be sorted so that they are listed according to their original ordering, and therefore

$$Q \ = \ \leqslant_{index}$$

the sorting being unnecessary for this particular example!

We hopefully check condition (3), but find it false. However, as P is naturally expressed as the intersection of two orderings, we turn to Lemma 3. Following the hint that P_Q is supposed to strongly resemble Q, we define $P_Q = Q$ and $P_R = \leqslant_{char}$. Conditions (1),(4)-(7) are trivially true, and thus the generic algorithm applies to this problem, which in this case is the same as that of Bird [2].

3.3. Box Stacking

This problem is concerned with the efficient stacking of a large number of cardboard boxes. Putting boxes inside each other makes good use of available space, and it is a reasonable goal to try and reduce the number of stacks to a minimum. We consider a simple two-dimensional version of box stacking, where we take into account the width (shortest side) and length (longest side) of each box, but do not worry about their heights. Two boxes of the same size will not fit inside each other, so we define

$$P = <_{width} \cap <_{length}$$

Note that this definition is sufficient, as there is no advantage to rotating the boxes: if it is the case that $length\ b_1 < width\ b_2$ and $width\ b_1 < length\ b_2$, then $b_1\ P\ b_2$ anyway.

We now have a minimum chains partitioning problem. One obvious possibility for Q is $\leqslant_{width}$. As condition (3) does not hold, but P is once again an intersection of orderings, we proceed with choosing the obvious $P_Q = <_{width}$ and $P_R = <_{length}$.

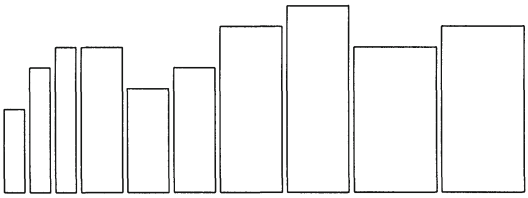
When checking the conditions, (5) and (6) are trivial to prove, but (7) is not true: for $b_1\ \neg P_R\ b_2$, we require that $length\ b_1 \geqslant length\ b_2$, but $b_1\ (Q \cap \neg P)\ b_2 \equiv width\ b_1 \leqslant width\ b_2\ \wedge\ (width\ b_1 \geqslant width\ b_2\ \vee\ length\ b_1 \geqslant length\ b_2)$. This is true if $width\ b_1 < width\ b_2$, but not if $width\ b_1 = width\ b_2$ and $length\ b_1 < length\ b_2$.

This is easily remedied by making Q more sophisticated, so that

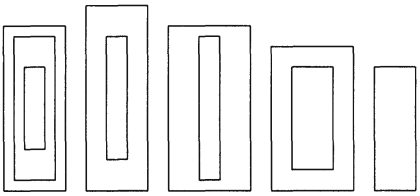
$$\begin{aligned} \forall\ b_1 b_2 \bullet b_1 Q' b_2 \equiv\ & width\ b_1 < width\ b_2 \\ & \vee (width\ b_1 = width\ b_2\ \wedge\ length\ b_1 \geqslant length\ b_2), \end{aligned}$$

making Q' a form of lexicographic ordering with $\leqslant_{width}$ and $\geqslant_{length}$. The conditions on P, Q', P_Q and P_R are now trivially easy to prove, and so the generic algorithm applies.

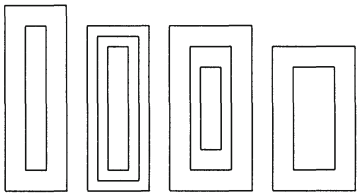
As an example, consider the list of boxes $[(1,4),(1,6),(1,7),(2,7)]+\!\!+$ $[(2,5),(2,6),(3,8),(3,9),(4,7),(4,8)]$:



Using the generic algorithm with the original Q and a stable sorting would have resulted in 5 chains being produced:



However, the use of the improved Q' gives an optimal 4 chains: $[[(1,7),(3,9)],[(1,6),(2,7),(3,8)],[(1,4),(2,6),(4,8)],[(2,5),(4,7)]]$:



An advantage of this algorithm is that it is practical to apply to real cardboard boxes, albeit that the sorted line of boxes might be very long!

4. Conclusions

This algorithm arose out of work with assorted greedy algorithms, as the "choose the leftmost longest chain" strategy is a greedy choice. The greedy algorithm for the Smallest Upravel problem was already known and provided the inspiration that the strategy might be more generally applicable. Investigation revealed that the Box Stacking and FSP minimum chain partitioning problems could also be solved in this way. It is interesting to note that the structure of the FSP seems to be fundamentally different from the Box Stacking and Smallest Upravel problems, with its ordering not expressed as an intersection, and with the stronger condition (3) satisfied. It remains to be seen how useful this generic algorithm is for other such minimum chain partitioning problems, given

that there exists a further problem not known to be solvable by this method, that of the three-dimensional box stacking problem, where

$$P = <_{width} \cap <_{length} \cap <_{height}$$

The author welcomes details of any more minimum chain partitioning problems.

In the meantime, it is a satisfactory result that the generic algorithm has not only unified three minimum chain partitioning problems, but also provided novel solutions for the Box Stacking and FSP problems.

Acknowledgments

Thanks are due to Gavin Lowe, who suggested this particular variant of the box stacking algorithm, and who helped to sort and stack an extremely large number of cardboard boxes. Finally, posthumous thanks to Menya Wolfe, a woman of great common sense, who pointed out that cutting the packing tape on the boxes and subsequently flattening them was a much better solution.

References

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/log\ n})$. *Information Processing Letters*, 37(4):237–240, 1991.

[2] R. S. Bird. The smallest upravel. *Science of Computer Programming*, 18:281–292, 1992.

[3] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[4] G. B. Dantzig and D. R. Fulkerson. Minimizing the number of tankers to meet a fixed schedule. *Naval Research Logistics Quarterly*, 1:217–222, 1954.

[5] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(2):161–166, 1950.

[6] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[7] U.I. Gupta, D.T. Lee, and J.Y. Leung. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers*, 28(11):807–810, November 1979.

[8] I. Gertsbakh and H.I. Stern. Minimal resources for fixed and variable job schedules. *Operations Research*, 26:68–85, 1978.

[9] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–230, December 1973.

[10] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, 1988.