The Knife Change Minimization Problem
Definition, Properties, Heuristsics

April 26, 1996

1 Introduction

We define formally the Knife Change Minimization Problem, we prove some properties which reduce the search space, and then describe some heuristsics.

At one of the last stages of the paper construction process customer widths have to be cut out of jumbo reels. For example, the widths 50,40,60,40, 30,50,50,50 and 60,40,40,40 may have to be cut out of three jumbo reels of width 200. The collections of indivdidual wdths (*e.g.* 50-40-60-40) are called *patterns*.

The order in which to consider the patterns (*i.e.* the route) can be arbitrary, and the order in which to cut each pattern is arbitrary as well. Each different solution involves a diffrerent number of knife changes, *e.g.* the solution from above involves 12 knife changes, whereas the solution 50-40-40-60, 5-4-4-4 and 50-50-50-30 involves only 7 knife changes. The objective is to find the solution with the minimal number of knife changes, or, because the search space is immense, to approximate such a solution.

We first give some auxiliary definitions describing operations on sequences, bags and sets. We then define formally the problem, the solution space and the cost function in terms of the above. We prove some properties which reduce the search space, and then we describe heuristsics.

1.1 Sequences

Sequences, the cardinality the inverse of a sequence, the difference of two sequences are defined as follows:

A sequence:
    type *seq* (α) == empty ++ α:: *seq* (α);
The number of elements in a sequence:
    fun *card* : *seq* (α) $\longrightarrow$ *int* ;
    — *card* empty = 0;
    — *card* x::xs = 1 + *card* (xs);
Appending an element, or appending a sequence
    fun *append* : *seq* (α) $\times$ α $\longrightarrow$ *seq* (α);
    — *append* empty a1 = a1::empty ;
    — *append* a::as a1 = a :: *append* (as, a1)
    fun *append* : *seq* (α) $\times$*seq* (α) $\longrightarrow$ *seq* (α);
    — *append* as empty = as;
    — *append* as b::bs = *append* (*append* (as, b), bs);
Prepending an element to sequence of sequences
    fun *prefix* : α $\times$ *seq* (*seq* (α)) $\longrightarrow$ *seq* (*seq* (α));

— *prefix* a1 empty = empty;
— *prefix* a1 as::ass = (a1::as) :: *prefix* (a1, ass);
Inverting a sequence
fun *inverse* : *seq* $(\alpha) \longrightarrow$ *seq* (α);
— *inverse* empty = 0;
— *inverse* a::as = *append* (*inverse* (as), a);
Whether an element appears in a sequence
fun *isIn* : *seq* $(\alpha) \times \alpha \longrightarrow$ *int*
— *isIn* empty x = 0;
— *isIn* y::ys x = (**if** x=y **then** 1 **else** 0) +*isIn* (ys,x);

1.2 Sets

Sets, the cardinality of a set, the difference and the union of two sets are defined as follows:

A set:
type *set* (α) == empty ++ α :: *set* (α);
The number of elements in a set:
fun *card* : *set* $(\alpha) \longrightarrow$ *int* ;
— *card* empty = 0;
— *card* x::xs = 1 + *card* (xs);
Whether an element appears in a set
fun *isIn* : *set* $(\alpha) \times \alpha \longrightarrow$ *bool*
— *isIn* x empty = false;
— *isIn* x y::ys = **if** x=y **then** true **else** *isIn* (ys, x);
The difference of two sets
fun *minus* : *set* $(\alpha) \times$ *set* $(\alpha) \longrightarrow$ *set* (α)
— *minus* empty xs = empty;
— *minus* x::xs ys = **if** *isIn* (x,ys) **then** *minus* (xs,ys) **else** x::*minus* (xs, ys);

1.3 Multisets or Bags

Multisets, or bags may contain an element more than once; they are defined as follows:

A bag:
type *bag* (α) == empty ++ ($\alpha \times$ *int*) :: *bag* (α); [1]
The number of elements in a bag:
fun *card* : *bag* $(\alpha) \longrightarrow$ *int* ;
— *card* empty = 0;
— *card* (x,i)::xs = i + *card* (xs);
Whether an element appears in a bag
fun *isIn* : *bag* $(\alpha) \times \alpha \longrightarrow$ *bool*
— *isIn* empty x= false;
— *isIn* (y,i)::ys x = **if** x=y **then** i **else** *isIn* (ys,x);
Removing an element, or another bag
fun *minus* : *bag* $(\alpha) \times \alpha \times$ int $\longrightarrow$ *bag* (α);
— *minus* empty a1 k = empty
— *minus* (a1,i)::as a1 k = **if** i-k¿0 **then** (a1,i-k)::as **else** as
— *minus* (a2,i)::as a1 k= (a2,i)::*add* (as,a1,k)
fun *minus* : *bag* $(\alpha) \times$ *bag* $(\alpha) \longrightarrow$ *bag* (α)

— *minus* as empty =xs;
— *minus* as (a1,k)::bs = *minus* (min(as,a1,k), bs)

1.4 Permutations

The permutations of the elements of a set:
fun *allPerms* : *set* (α) $\longrightarrow$ *set* (*seq* (α))
— *allPerms* s = { t | and $\forall$a$\in$ α: isIn(t,a)=1 iff isIn(s,a) }
Notice that *card* (*allPerms* (s))=*card* (s)!
The permutations of the elements of a bag:
fun *allPerms* : *bag* (α) $\longrightarrow$ *set* (*seq* (α))
— *allPerms* b = { t | and $\forall$a$\in$ α: isIn(t,a)=isIn(b,a) }
Notice that for a bag=(a1:i1)::...::(an::in)::empty, *card* (*allPerms* (bag))=(i1+i2+....+in)!/(i1!*i2!*...in!)

1.5 The Problem

We now define the problem:

type *Width* = *int* ;
type *Pattern* = *bag* (*Width*);
type *Problem* = *set* (*Pattern*);
Notice, that a pattern is a *bag* of widths, *i.e.* repetition is possible.
The problem is tepresentedby a set of patterns; if there is repetiton, this can be detected, and removed.

type *CutInstr* = *seq* (*Width*);
type *Solution* = *seq* (*CutInstr*);
A particular solution consists of a sequence of Cut Instructions.
Cut Instructions express in which order to cut the various items in a pattern.

The solution space is described by:
fun *allSolutions* : *Problem* $\longrightarrow$ *set* (*Solution*);
— *allSolutions* problem = *allCutInstrs* (*allRoutes* (problem));

A route describes an order in which to consider the patterns
type *Route* = *seq* (*Pattern*) ;
Any permutation of the patterns in the problem is a possible route
fun *allRoutes* : *Problem* 0 $\longrightarrow$ *set* (*Route*);
— *allRoutes* pr = { r | r $\in$ *allPerms* (pr) }
The cut instructions corresponding to one pattern are all possible permutations of the widths in this pattern
fun *allCutInstrs* : *Pattern* $\longrightarrow$ *set* (*CutInstr*);
— *allCutInstrs* pa = { c | c $\in$ permutations(pa) }
For a given route the sequence of cut instruction consists of a cut instruction per pattern in the order they appear in the route
fun *allCutInstrs* : *Route* $\longrightarrow$ *set* (*Solution*);
— *allCutInstrs* $\mathtt{pa}_1$::$\mathtt{pa}_2$ ... ::$\mathtt{pa}_n$ = { $\mathtt{c}_1$::$\mathtt{c}_2$ ... ::$\mathtt{c}_n$ | $\mathtt{c}_i$ $\in$ *allCutInstrs* ($\mathtt{pa}_i$), for i=1,..n };

1.6 The Objective, and Cost of a Solution

The aim of the Knife Change Minimization Project is to find a solution with minimal *cost* , *i.e.* for a given pr∈*Problem* , to find a s∈*allSolutions* (pr), such that:

∀ s'∈*allSolutions* (pr): *cost* (s) ≤ *cost* (s')

The cost of a solution is defined as the number of necessary knife (re-)positionings.

fun *cost* : *Solution* ⟶ *int* ;
— *cost* empty = 0;
— *cost* p::ps = *card* (p) + *costAux* (ps,p)

The *cost* of one solution

fun *cost* : *Solution* ⟶ *int* ;
— *cost* empty = 0;
— *cost* p::ps = *card* (p) + *costAux* (ps,p)
fun *costAux* : *Solution* × *CutInstr* ⟶ *int* ;
— *costAux* empty p = 0;
— *costAux* p1::ps p2 = *knifeChanges* (p1, p2) + *costAux* (ps, p2);

The number of knife changes necessary from one cut instruction to another

fun *knifeChanges* : *CutInstr* × *CutInstr* ⟶ *int* ;
— *knifeChanges* p1 p2 = *card* (*minus* (*knifePosns* (p2), *knifePosns* (p1)));

The positions at which knives need to be placed in order to cut a cut instruction:

type *Positions* = *seq* (*Width*);
fun *knifePosns* : *CutInstr* ⟶ *Positions* ;
— *knifePosns* p = *knifePosnsAux* p 0 **where**
fun *knifePosnsAux* : *CutInstr* × *Width* ⟶ *Positions* ;
— *knifePosnsAux* empty k = empty;
— *knifePosnsAux* i::is k = (i+k)::*knifePosnsAux* (is, i+k);

2 Properties

2.1 Inverse-Lemma

The following lemma says that a solution and its inverse have the same cost. This cuts the search space by a half.

Lemma: For any s ∈ *Solution* :

$$cost\ (\ s\ ) = cost\ (\ inverse\ (\ s\ ))$$

Proof:
A. Observe that for a solution s=i_1::i_2::...i_n:

$$cost\ (s) = card\ (l_1) + card\ (minus\ (l_2, l_1)) + \ldots\ card\ (minus\ (l_n, l_{n-1}))$$

where l_j=*knifePosns* (i_j). The above holds by application of the definition of *cost* , and also, because for any cut instruction i, *card* (i)=*card* (*knifePosns* (i)).
B. Also, observe that for any two sequences l, l':

$$card\ (l) + card\ (minus\ (l', l)) = card\ (l') + card\ (minus\ (l, l'))$$

which can be proven by induction over the number of elements in sequence l'. (Basically, both sides of the expression represent the cardinality of l and l'.)
C: We now show, that for any sequence of sequences $l_1, \ldots l_n$:

$$\mathit{card}\ (l_1)+\mathit{card}\ (\mathit{minus}\ (l_2,l_1))+\ldots\mathit{card}\ (\mathit{minus}\ (l_n,l_{n-1}))= \\ \mathit{card}\ (l_n)+\mathit{card}\ (\mathit{minus}\ (l_{n-1},l_n))\ldots\mathit{card}\ (\mathit{minus}\ (l_1,l_2))$$

which we can prove by induction over the number n.
Base case: n=1, C vacuuosly true.
Induction step: from n to n+1:

$$\begin{array}{ll}
\mathit{card}\ (l_1)+\mathit{card}\ (\mathit{minus}\ (l_2,l_1))+\ldots\mathit{card}\ (\mathit{minus}\ (l_{n+1},l_n)) = & \text{(expand)} \\
\mathit{card}\ (l_1)+\mathit{card}\ (\mathit{minus}\ (l_2,l_1))+\ldots\mathit{card}\ (\mathit{minus}\ (l_n,l_{n-1}))\ +\ \mathit{card}\ (\mathit{minus}\ (l_{n+1},l_n)) = & \text{(I.H.)} \\
\mathit{card}\ (l_n)+\mathit{card}\ (\mathit{minus}\ (l_{n-1},l_n))+\ldots\mathit{card}\ (\mathit{minus}\ (l_1,l_2))\ +\mathit{card}\ (\mathit{minus}\ (l_{n+1},l_n)) = & \text{(rearrange)} \\
\mathit{card}\ (\mathit{minus}\ (l_{n+1},l_n))+\mathit{card}\ (l_n)\ +\mathit{card}\ (\mathit{minus}\ (l_{n-1},l_n))+\ldots\mathit{card}\ (\mathit{minus}\ (l_1,l_2)) = & \text{(B)} \\
\mathit{card}\ (l_{n+1})+\ \mathit{card}\ (\mathit{minus}\ (ln,l_{n+1})\ +\mathit{card}\ (\mathit{minus}\ (l_{n-1},l_n))+\ldots\mathit{card}\ (\mathit{minus}\ (l_1,l_2)) = & \text{(fold)} \\
\mathit{card}\ (l_{n+1})+\mathit{card}\ (\mathit{minus}\ (l_{n+1},l_n))+\ldots\mathit{card}\ (\mathit{minus}\ (l_1,l_2)). & \text{q.e.d}
\end{array}$$

D: Combining A and C:

$$\mathit{cost}\ (\ s\ ) = \mathit{cost}\ (\ \mathit{inverse}\ (\ s\ ))$$

2.2 Common Item Property

Lemma: For any l_1, l_2, l_3, l_4 $\in$ *CutInstr* , there exist l_5, l_6 $\in$ *CutInstr* , with $l_5 \in$Perms(l_1 ++ i::l_3), $l_6 \in$Perms(l_2++i::l_4) such that:

$$\mathit{cost}\ (\ i::l_5\ ++\ i::l_6\ ) \leq \mathit{cost}\ (\ l_1++i::l_3\ ++\ l_2++i::l_4\ )$$

Proof: By case analysis over ....

2.3 Shift Property

The following lemma says that there is a simple way of finding the solution with the best sost, when considering the m solutions that can be obteined by shifting an original solution

Lemma: Consider any solution s=s=i_1::i_2::...i_m, and s_j=shift^j(s) for j $\in$0..m-1, For the k$\in$1..m, such that $\mathit{card}\ (i_k)\text{-}\mathit{knifeChanges}\ (i_{k-1}i_k = \min_{j\in 0..m-1}\mathit{card}\ (i_j)+\mathit{knifeChanges}\ (i_{j-1}i_j)$

$$\mathit{cost}\ (s_k)= \min_{j\in 0..m-1}\mathit{cost}\ (s_j\ )$$

where the operations +, - are modulus m, and ++ is an infix notation of the *append* operator.

Proof: Let us define M=KnifeChanges(i_1,i_2)+...KnifeChanges(i_{n-1},i_n). Then $\mathit{cost}\ (s_j\ )$=M+$\mathit{card}\ (i_j)$-$\mathit{knifeChanges}\ (i_{j-1}i_j)$, which proves the conjecture.

3 Heuristsics

The heuristsics will try to create initial good solutions which can be used by the genetic algorithms as the inital population. A heuristic will take a problem a return a solution. Intermediate heuristics produce the route out of a aproblem and others produce a solution out of a given route.

type *Heuristic* **type** *ProblemHeuristic* = *Heuristic* *Solution* → *Route* → *Route* → *Solution*

3.1 Most Common Width

This heuritic finds w, the width that appears in most patterns. Then it finds all patterns that contain this width (We repeat this recursively, until there ar no coomon widths This is based on the common item property. It is a kind of depth first, greedy heuristic.

fun *heuristic1* : *Problem* $\longrightarrow$ *Solution* ;
— *heuristic1* problem = *prefix* (w , *heuristic1* (*minus* (problem1,w)))
:: *heuristic1* (problem2);
where w such that: $\forall$ w' *nrAppears* (w, problem) $\geq$ *nrAppears* (w', problem)
problem=problem1::problem2 and $\forall$p isIn(problem1,p) iff isIn(p,w)

Furthermore, the function *minus* removes from the problem the wirdth w:
fun *minus* : *set* (*bag* (α)) $\times$ α $\longrightarrow$ *set* (*bag* (α));
— *minus* empty a = empty;
— *minus* b1::bs a = *minus* (b1,a)::*minus* (bs,a);

and the function *nrAppears* counts the number of patterns in which a width appears:
fun *nrAppears* : *set* (*bag* (α)) $\times$ α $\longrightarrow$ *int* ;
— *nrAppears* empty a = 0;
— *nrAppears* b1::bs a = (**if** *isIn* (b1,a) **then** 1 **else** 0) + *nrAppears* (bs,a);

For example, the following solution might be the result of *heuristic1* :

300 - 250 - 350 - 100
300 - 250 - 350
300 - 250 - 350
300 - 140 - 400
150 - 350 - 350
150 - 200
150 - 400 - 100

Notice, that 350 appears as often as 300, but but 300 was chosen as ythe first most common width (the above defintion is non-determinsitc).

The following example of an application of this heuristic:
35 - 20 - 20 - 70 - 55 - 30
35 - 100 - 60 - 20
35 - 92 - 55 - 25
45 - 20 - 20 - 70 - 55 - 30
demonstrates its disadvantages, namely, the solution
20 - 20 - 70 - 55 - 30 - 35
20 - 20 - 70 - 55 - 30 - 45
35 - 100 - 60 - 20
35 - 92 - 55 - 25
would have been much better.

3.2 Largest Common Set

This heurirtic is "breadth first": it tries to establish the largest block of widths common to two neighbouting patterns. The distance of a pair of patterns is the number of widths appearing in both, divided by the

fun *heuristic2* : *Problem* $\longrightarrow$ *Solution* ;
— *heuristic2* problem = *cutInstrHeuristic* (*routeHeuristic* (problem));

for appropriate functions:

fun *cutInstrHeuristic* : *CutInstrHeuristic* ;
fun *routeHeuristic* : *RouteHeuristic* ;

The distance of two patterns counts the number of items which are not common to the two of them:

fun *dist* : *Pattern* $\times$ *Pattern* $\longrightarrow$ real;
— *dist* pa1 pa2 = *card* (*add* (pa1,pa2)) - *card* (*intersection* (pa1,pa2));

This distance can be used for the definition of a route heuristic:

— *routeHeuristic* = attempt to solve a TSP using *dist* as a distance measure for the patterns
several heuristics possible, nearest neighbour good first approximation
— *cutInstrHeuristic* route = *prefix* (w , *cutInstrHeuristic* (minus(route1,w)))
:: *cutInstrHeuristic* (route2);
where w, route1, route2 such that:
route=*append* (route1,route2) **and** $\forall$ patterns p, *isIn* (route1,p): *appersIn* (w,route1)

3.3 Hybrid

fun *heuristic3* : *Problem* $\longrightarrow$ *Solution* ;
— *heuristic2* problem = append(*heuristic1* (problem1), *heuristic3* (problem2));
where
pronlem=*add* (problem1,problem2)
$\forall$ patterns p1,p2, *totalWidth* (p1)=*totalWidth* (p2)