The Knife Change Minimization Problem
Definition, Properties, Heuristics

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 heuristics.

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 and 60,40,40,40 may have to be cut out of three jumbo reels of width 200. The collections of individual widths (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 different 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 heuristics.

1.1 Sequences

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

A sequence:

\[ \text{type seq} \ (\alpha) = \text{empty} ++ \alpha:: \text{seq} \ (\alpha); \]

The number of elements in a sequence:

\[ \text{fun card} : \text{seq} \ (\alpha) \rightarrow \text{int}; \]
- \[ \text{card empty} = 0; \]
- \[ \text{card} \ x::xs = 1 + \text{card} \ (xs); \]

Appending an element, or appending a sequence

\[ \text{fun append} : \text{seq} \ (\alpha) \times \alpha \rightarrow \text{seq} \ (\alpha); \]
- \[ \text{append} \ \text{empty} \ a1 = a1::\text{empty}; \]
- \[ \text{append} \ a::as \ a1 = a :: \text{append} \ (\ as, a1) \]

\[ \text{fun append} : \text{seq} \ (\alpha) \times \text{seq} \ (\alpha) \rightarrow \text{seq} \ (\alpha); \]
- \[ \text{append} \ as \ \text{empty} = as; \]
- \[ \text{append} \ as \ b::bs = \text{append} \ (\ \text{append} \ (\ as, b), bs); \]

Prepending an element to sequence of sequences

\[ \text{fun prefix} : \alpha \times \text{seq} \ (\text{seq} \ (\alpha)) \rightarrow \text{seq} \ (\text{seq} \ (\alpha)); \]
Inverting a sequence

```haskell```
fun inverse : seq (α) → seq (α);
- inverse empty = 0;
- inverse a::as = append (inverse (as), a);
```

Whether an element appears in a sequence

```haskell```
fun isIn : seq (α) × α → 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:

```haskell```
type set (α) == empty ++ α :: set (α);
```

The number of elements in a set:

```haskell```
fun card : set (α) → int ;
- card empty = 0;
- card x::xs = 1 + card (xs);
```

Whether an element appears in a set

```haskell```
fun isIn : set (α) × α → bool
- isIn x empty = false;
- isIn x y::ys = if x=y then true else isIn (ys, x);
```

The difference of two sets

```haskell```
fun minus : set (α) × set (α) → 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:

```haskell```
type bag (α) == empty ++ (α × int ) :: bag (α); ^{1}
```

The number of elements in a bag:

```haskell```
fun card : bag (α) → int ;
- card empty = 0;
- card (x,i)::xs = i + card (xs);
```

Whether an element appears in a bag

```haskell```
fun isIn : bag (α) × α → 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

```haskell```
fun minus : bag (α) × α × int → 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)
```

```haskell```
fun minus : bag (α) × bag (α) → 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:

\[
\text{fun allPerms : set } (\alpha) \rightarrow \text{set } (\text{seq } (\alpha))
\]

— allPerms s = \{ t | and \forall \alpha : \text{isIn}(t,\alpha)=1 \iff \text{isIn}(s,\alpha) \} 

Notice that card (allPerms (s))=card (s)!

The permutations of the elements of a bag:

\[
\text{fun allPerms : bag } (\alpha) \rightarrow \text{set } (\text{seq } (\alpha))
\]

— allPerms b = \{ t | and \forall \alpha : \text{isIn}(t,\alpha)=\text{isIn}(b,\alpha) \} 

Notice that for a bag=(a1;i1):...:(an;iin)::empty, card (allPerms (bag))=(i1+i2+...+in)!/(i1!*i2!*...*in!)

1.5 The Problem

We now define the problem:

\[
\text{type Width = int ;}
\]
\[
\text{type Pattern = bag } (\text{Width });
\]
\[
\text{type Problem = set } (\text{Pattern });
\]

Notice, that a pattern is a bag of widths, i.e. repetition is possible.

The problem is represented by a set of patterns; if there is repetition, this can be detected, and removed.

\[
\text{type CutInstr = seq } (\text{Width });
\]
\[
\text{type Solution = seq } (\text{CutInstr });
\]

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:

\[
\text{fun allSolutions : Problem } \rightarrow \text{set } (\text{Solution });
\]

— allSolutions problem = allCutInstrs (allRoutes (problem));

A route describes an order in which to consider the patterns

\[
\text{type Route = seq } (\text{Pattern });
\]

Any permutation of the patterns in the problem is a possible route

\[
\text{fun allRoutes : Problem } 0 \rightarrow \text{set } (\text{Route });
\]

— allRoutes pr = \{ r | r \in \text{allPerms } (pr) \}

The cut instructions corresponding to one pattern are all possible permutations of the widths in this pattern

\[
\text{fun allCutInstrs : Pattern } \rightarrow \text{set } (\text{CutInstr });
\]

— allCutInstrs pa = \{ c | c \in \text{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

\[
\text{fun allCutInstrs : Route } \rightarrow \text{set } (\text{Solution });
\]

— allCutInstrs pa1::pa2 ... ::pa_n = \{ c_1::c_2 ... ::c_n | c_i \in \text{allCutInstrs } (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 Problem, to find a \( s \in \text{allSolutions} \) (pr), such that:

\[ \forall s' \in \text{allSolutions} \text{ (pr): } cost (s) \leq cost (s') \]

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

\[
\text{fun cost: Solution \rightarrow int;}
\]
\[
\text{cost empty } = 0;
\]
\[
\text{cost p::ps = card (p) + costAux (ps,p)}
\]

The cost of one solution

\[
\text{fun cost: Solution \rightarrow int;}
\]
\[
\text{cost empty } = 0;
\]
\[
\text{cost p::ps = card (p) + costAux (ps,p)}
\]

\[
\text{fun costAux: Solution \times CutInstr \rightarrow int;}
\]
\[
\text{costAux empty p } = 0;
\]
\[
\text{costAux p1::ps p2 = knifeChanges ( p1, p2) + costAux ( ps, p2 );}
\]

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

\[
\text{fun knifeChanges: CutInstr \times CutInstr \rightarrow int;}
\]
\[
\text{knifeChanges p1 p2 = card (minus ( knifePosns ( p2), knifePosns ( p1) ));}
\]

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

\[
\text{type Positions } = \text{ seq ( Width );}
\]
\[
\text{fun knifePosns: CutInstr \rightarrow Positions ;}
\]
\[
\text{knifePosns p = knifePosnsAux p 0 where}
\]
\[
\text{fun knifePosnsAux: CutInstr \times Width \rightarrow Positions ;}
\]
\[
\text{knifePosnsAux empty k } = \text{ empty;}
\]
\[
\text{knifePosnsAux i::is k } = (i+k)::\text{knifePosnsAux ( is, i+k )};
\]

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 \( s \in \text{Solution} \):

\[ cost ( s ) = cost ( \text{inverse} ( s ) ) \]

**Proof:**

A. Observe that for a solution \( s = i_1::i_2::...::i_n \):

\[
\text{cost (s)} = \text{card (l_1)} + \text{card (minus (l_2,l_1))} + ... \text{ card (minus (l_n,l_{n-1}))}
\]

where \( l_j = \text{knifePosns} (i_j) \). The above holds by application of the definition of \( cost \), and also, because for any cut instruction \( i \), \( \text{card (i)} = \text{card (knifePosns (i))} \).

B. Also, observe that for any two sequences \( l, l' \):

\[
\text{card (l)} + \text{card (minus (l',l))} = \text{card (l')} + \text{card (minus (l,l'))}
\]
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 \( l_1, \ldots, l_n \):

\[
\text{card} \ (l_1) + \text{card} \ (\text{minus} \ (l_2, l_1)) + \ldots + \text{card} \ (\text{minus} \ (l_n, l_{n-1})) = \\
\text{card} \ (l_n) + \text{card} \ (\text{minus} \ (l_{n-1}, l_n)) + \ldots + \text{card} \ (\text{minus} \ (l_1, l_2))
\]

which we can prove by induction over the number \( n \).

Base case: \( n = 1 \), C vacuously true.

Induction step: from \( n \) to \( n+1 \):

\[
\begin{align*}
\text{card} \ (l_1) + & \text{card} \ (\text{minus} \ (l_2, l_1)) + \ldots + \text{card} \ (\text{minus} \ (l_n, l_{n+1})) = & \text{ (expand)} \\
\text{card} \ (l_1) + & \text{card} \ (\text{minus} \ (l_2, l_1)) + \ldots + \text{card} \ (\text{minus} \ (l_n, l_{n+1})) + \text{card} \ (\text{minus} \ (l_{n+1}, l_n)) = & \text{ (I.H.)} \\
\text{card} \ (l_n) + & \text{card} \ (\text{minus} \ (l_{n+1}, l_n)) + \ldots + \text{card} \ (\text{minus} \ (l_1, l_2)) = & \text{ (rearrange)} \\
\text{card} \ (\text{minus} \ (l_{n+1}, l_n)) + & \text{card} \ (l_n) + \text{card} \ (\text{minus} \ (l_{n-1}, l_{n+1})) + \ldots + \text{card} \ (\text{minus} \ (l_1, l_2)) = & \text{ (B)} \\
\text{card} \ (l_{n+1}) + & \text{card} \ (\text{minus} \ (l_n, l_{n+1})) + \text{card} \ (\text{minus} \ (l_{n-1}, l_{n+1})) + \ldots + \text{card} \ (\text{minus} \ (l_1, l_2)) = & \text{ (fold)} \\
\text{card} \ (l_{n+1}) + & \text{card} \ (\text{minus} \ (l_{n+1}, l_n)) + \ldots + \text{card} \ (\text{minus} \ (l_1, l_2)). & \text{ q.e.d}
\end{align*}
\]

D: Combining A and C:

\[
\cost \ (s) = \cost \ (\text{inverse} \ (s))
\]

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 \( l_5 \in \text{Perms}(l_1 ++ i::l_3) \), \( l_6 \in \text{Perms}(l_2 ++ i::l_4) \) such that:

\[
\cost \ (i::l_5 ++ i::l_6) \leq \cost \ (l_1 ++ i::l_3 ++ l_2 ++ i::l_4)
\]

**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 cost, when considering the \( m \) solutions that can be obtained by shifting an original solution.

**Lemma:** Consider any solution \( s = s_1::s_2::\ldots::s_m \), and \( s_j = \text{shift}^j(s) \) for \( j \in 0..m-1 \). For the \( k \in 1..m \), such that \( \text{card} \ (i_k) \)-\text{knifeChanges} \ (i_{k-1}i_k = \min_{j \in 0..m-1} \text{card} \ (i_j) + \text{knifeChanges} \ (i_{j-1}i_j) \)

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

where the operations \(+, -\) are modulus \( m \), and \( ++ \) is an infix notation of the \text{append} operator.

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

3 Heuristics

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

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

\[
\text{fun} \, \text{heuristic1} : \text{Problem} \rightarrow \text{Solution} ;
\]

\[
\begin{align*}
&\text{heuristic1} \text{ problem} = \text{prefix} ( w , \text{heuristic1} (\text{minus} (\text{problem1},w))) \\
&\text{heuristic1} (\text{problem2}) ;
\end{align*}
\]

\[
\text{where} \ w \ \text{such that: } \forall \ w' \ \text{nrAppears} (w, \text{problem}) \geq \text{nrAppears} (w', \text{problem}) \text{ problem} = \text{problem1} : \text{problem2} \text{ and } \forall p \ \text{isIn}(\text{problem1},p) \text{ iff } \text{isIn}(p,w)
\]

Furthermore, the function \( \text{minus} \) removes from the problem the width \( w \):

\[
\text{fun} \, \text{minus} : \text{set} (\text{bag} (\alpha)) \times \alpha \rightarrow \text{set} (\text{bag} (\alpha));
\]

\[
\begin{align*}
&\text{minus} \ \text{empty} a = \text{empty}; \\
&\text{minus} \ b1::bs a = \text{minus} (b1,a)::\text{minus} (bs,a);
\end{align*}
\]

and the function \( \text{nrAppears} \) counts the number of patterns in which a width appears:

\[
\text{fun} \, \text{nrAppears} : \text{set} (\text{bag} (\alpha)) \times \alpha \rightarrow \text{int} ;
\]

\[
\begin{align*}
&\text{nrAppears} \ \text{empty} a = 0; \\
&\text{nrAppears} \ b1::bs a = (\text{if } \text{isIn} (b1,a) \text{ then } 1 \text{ else } 0) + \text{nrAppears} (bs,a);
\end{align*}
\]

For example, the following solution might be the result of \( \text{heuristic1} \):

\[
\begin{align*}
300 - 250 - 350 - 100 \\
300 - 250 - 350 \\
300 - 250 - 350 \\
300 - 140 - 400 \\
150 - 350 - 350 \\
150 - 200 \\
150 - 400 - 100
\end{align*}
\]

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

The following example of an application of this heuristic:

\[
\begin{align*}
35 - 20 - 20 - 70 - 55 - 30 \\
35 - 100 - 60 - 20 \\
35 - 92 - 55 - 25 \\
45 - 20 - 20 - 70 - 55 - 30
\end{align*}
\]

demonstrates its disadvantages, namely, the solution

\[
\begin{align*}
20 - 20 - 70 - 55 - 30 - 35 \\
20 - 20 - 70 - 55 - 30 - 45 \\
35 - 100 - 60 - 20 \\
35 - 92 - 55 - 25
\end{align*}
\]

would have been much better.

3.2 Largest Common Set

This heuristic is "breadth first": it tries to establish the largest block of widths common to two neighbouring patterns. The distance of a pair of patterns is the number of widths appearing in both, divided by the
fun heuristic2 : Problem → 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 × Pattern → 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 ∀ patterns p, isIn (route1,p): appersIn (w,route1)

3.3 Hybrid

fun heuristic3 : Problem → Solution ;
— heuristic3 problem = append( heuristic1 (problem1), heuristic3 (problem2));

where
  problem = add (problem1,problem2)
∀ patterns p1,p2, totalWidth (p1)=totalWidth (p2)