NOVEMBER 1996

WRL
Research Report 96/1

Optimization in Permutation Spaces

Silvio Turrini

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

Optimization in Permutation Spaces

Silvio Turrini

November 1996

d|i|g|i|t|a|l **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

Many optimization problems find a natural mapping in permutation spaces where dedicated algorithms can be used during the optimization process. Unfortunately, some of the best and most effective techniques currently used can only be applied to vectors (cartesian) spaces, where a concept of distance between different objects can be easily defined. Examples of such techniques go from simplest deepest descent hill climbers and the more sophisticated conjugate gradient methods used in continuous spaces, to dynanic hill climbers or Genetic algorithms (GAs) used in many large combinatorial problems. This paper describes a general method that allows the best optimization techniques used in vector spaces to be applied to all order based problems whose domain is a permutation space. It will also be shown how this method can be applied to a real world problem, the optimal placement of interconnected cells (modules) on a chip, in order to minimize the total length of their connections. For this problem a dynamic hill climber has been used as the optimization engine, but other techniques that work in a multidimensional vector space can be applied as well.

Cartesian and Permutation Spaces

Optimization problems where the domains of the parameters to be optimized take on sets of independent values are said to belong to cartesian, or vector spaces. Problems with domains that are permutations of elements are said to belong to permutation spaces. In the former case the values that the parameters can take are *independent* from each other and the function to be optimized can geometrically be represented in a multidimensional space with as many dimensions as there are parameters. In the latter case the order of the elements which constitutes the n-tupla of values is what differentiate one input from another and the value of any parameter at a given position in the n-tupla is clearly *dependent* on all the others.

Example 1 :

A two variable function to optimize (*cartesian continuous space*) :

$\mathbf{F(x, y) = (x - y)^4 - (x - y)^2}$ where $\mathbf{x \in [0 .. 5], y \in [1 .. 4]}$ [see Fig.1]

A three variable function described by a permutation (*discrete permutation space*) :

$\mathbf{Q(x, y, z) = x \times P(x) + y \times P(y) + z \times P(z)}$

where $\mathbf{x \in [1 .. 3]}$ and $\mathbf{P(x) =}$ position of $\mathbf{x}$ in the permutation.

[see Fig.2]

F(x, y)

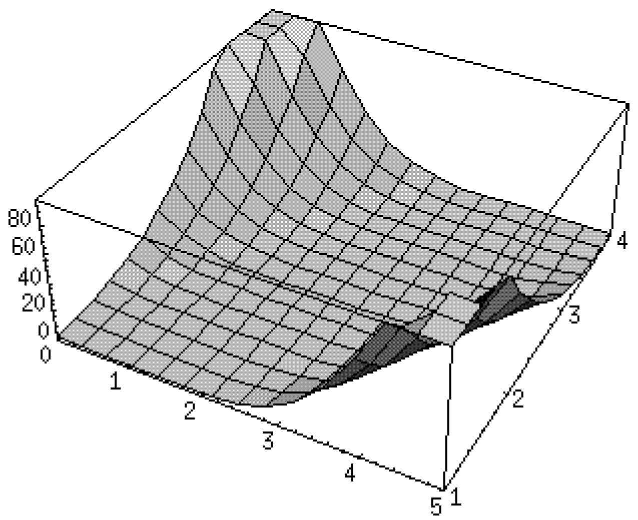

Fig 1 : values of **F(x, y)**

Q(x, y, z) :

Q(1, 2, 3) = 1 + 4 + 9 = 14; Q(1, 3, 2) = 1 + 6 + 6 = 13; Q(2, 1, 3) = 2 + 2 + 9 = 13;
Q(2, 3, 1) = 2 + 6 + 3 = 11; Q(3, 1, 2) = 3 + 2 + 6 = 11; Q(3, 2, 1) = 3 + 4 + 3 = 10;

Q(1, 2, 2), Q(1,1, 3), Q(3, 3 ,2) ... etc. are all **non-valid permutations**

Fig 2 : values of **Q(x, y, z)**

Workarounds when dealing with permutations

Regardless of which technique is used, dealing with vectors of parameters that must be optimized it is easier than working with their permutations. When iterative algorithms are used a few workarounds can be applied to overcome the problem :

Penalty functions (it is a very popular technique used with genetic algorithms) where an input sequence is penalized the more it is "far" from a legal permutation.

Example 2 : Suppose we want to *minimize* a given objective function ***F(x)*** whose parameters can take integer values in the range : ***1 ... n..***
Moreover, say that ***F(x)*** takes values on the range : ***min ... max.***
A possible penalty function ***p(x)*** could be :

$$p(x) = 1 + \textit{number of elements with the same value} \times \textbf{min}$$

with a new modified objective function :

$$F^*(x) = p(x) \times F(x)$$

so that all legal permutations still have the old values and illegal ones are increasingly penalized according to the number of "*wrong*" elements in the sequence.

Only "legal" input values can be generated during the iterative process.

For instance, in GAs special crossover and mutation operators are developed, or in simulated annealing techniques only swapping is allowed between the elements of a permutation.

Example 3 : In GAs a quite popular crossover operator is the so called Partial Matched crossover (PMX) first defined by Goldberg [Gold89]. The two chromosomes (parents) are aligned and two crossing sites are randomly chosen along them. These two points define a *matching section* which identifies the genes that will be exchanged (swapped) in each of the parent. In the example on the next page [see Fig. 3] the following elements will be swapped : $\mathbf{2 \leftrightarrow 2}$, $\mathbf{4 \leftrightarrow 7}$, $\mathbf{7 \leftrightarrow 4}$, $\mathbf{8 \leftrightarrow 6}$

1 2 4 7 8 3 5 6 *string representing 1st parent permutation*

5 2 7 4 6 3 1 8 *string representing 2nd parent permutation*

random crossover sites

Elements in columns between crossover sites are swapped

1 2 7 4 6 3 5 8 *string representing 1st child permutation*

5 2 4 7 8 3 1 6 *string representing 2nd child permutation*

random crossover sites

The new children are still legal permutations.

Fig. 3 : Partial Matched Crossover

This is an easy to implement order-based crossover, unfortunately the *semantics* of the operation and its effectiveness depend on the problem; in many cases this operator can be totally inadequate.

Both methods offer advantages and disadvantages, but most of the time they "*obscure*" the problem by adding complexity to the algorithm and decreasing its effectiveness.

Transformed Spaces

The concept of analytical transformation has been a very successful one and it has been applied to many difficult problems in physics and engineering as well. A typical example is the Fourier Transform which allows a electric signal to be "transformed" from a time domain into a frequency domain [see Fig. 4]. Some of the most complex operations that must be applied to signals, become very simple in the corresponding space, so that they can be efficiently carried out after the conversion has taken place . Convolution for example is a complex operation in the time domain which has a correspondent simple one in the frequency domain. Once all the work has been done in the transformed space, by using an inverse transformation the modified signal is converted back to the time domain. The key to this technique is how fast the transformation really is. If most of the computation is going in the forward and back conversion of the signal, no much is gained by using this approach. In the case of the Fourier transform, there was a real breakthrough when Cooley and Tukey [CoTuk65] discover a new algorithm with complexity $O(n \log n)$ instead of $O(n^2)$ of its more obvious implementation. With a much faster transform (FFT) the techniques used in signal analysis and the wonderful things that now signal processing can do really blossomed and we can certainly say that without such a fast transform this area would not have enjoyed the incredible growth we see today.

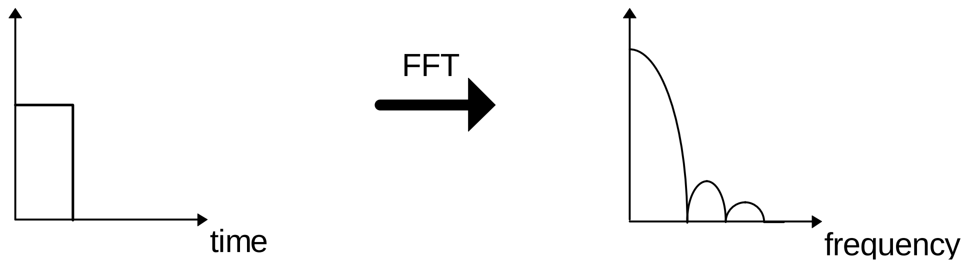

Fig. 4 : Fast Fourier Transform

We might think that such a clever idea could also be used in the permutation space. Permutations could be *mapped* into a vector space, with a *O*(*n log n*) transformation, so that linear operations can be carried out on the corresponding vectors. To go back to the original permutations another fast transformation must also be available. Surprisingly enough, a mathematical object that fulfills our needs has already been described and an algorithm with *O*(*n log n*) complexity already been suggested. For historical reasons this transformation takes the name of ***Inversion Table*** and its description can be found in Knuth's book [Knuth73] but as far as we know it has never been used for any of purposes discussed here. In the book the only use of the inversion table has been as a mathematical tool to prove theorems and properties of permutations. Being the permutation space mapped into a vector space would give us a way to *measure* distances between different sample points in the search space, which is harder to do in the original permutation domain. One simple way this information can be used during the search for optimal points is to identify interesting areas that look promising and avoid the less successful ones. Almost all iterative methods that operate on large search spaces use some *heuristics* to "guess" where the next good point to be sampled will be, based on some measure that correlates previous samples. If our algorithms can operate in a vector space there is already a well developed body of theories and practical solutions that can be applied to our order-based problem directly. This is clearly not the only way this transformation could be used. If the objective function we are optimizing has some special properties about its global maximum and minimum and requires operations that have a simple mapping in the transformed space, it is also conceivable to operate directly in the linear space and go back to the permutation domain only after the optimization is finished.

The Inversion Table

One way of defining the inversion table is :
given a permutation of **n** integers $\{ a_1, a_2 \dots a_n \}$ from the ordered set $\{ 1, 2 \dots n \}$, its inversion table $\{ b_1, b_2 \dots b_n \}$ is obtained by letting b_j be the number of elements to the left of element **j** that are greater than **j.**

Example 4 :

> the permutation **5 6 1 3 2 4 8 7** has the inversion table **2 3 2 2 0 0 1 0,** because to the left of element **1** there are two elements, **5** and **6,** to the left of element **2** there are three elements, **5, 6** and **3** and so on**.** Notice that other simple definitions are possible, such as counting all the elements on the right of **j**, or using *less than* instead of *greater than* for the comparison. By this definition the last value *must* always be **0**, therefore only ***n - 1*** components of the generated vector are meaningful.

The mathematical expression of what has just been said is :

$$b_{a_j} = \sum_{i=1}^{i=j} \textit{if } (a_i > a_j)\ 1;\ \textit{else } 0; \qquad \textbf{Eq. 1}$$

where **:** $0 <= b_1 < n-1, \quad 0 <= b_2 < n - 2 \ \dots\ b_n = 0;$

Every b_j can take values from a range than depends on its index $j;\ b_j \in \{0 .. n - j\}$.
For the permutation in the example :

$$b_1 \in \{0 .. 7\},\ b_2 \in \{0 .. 6\} \ \dots.\ b_7 \in \{0, 1\},\ b_8 \in \{0\}$$

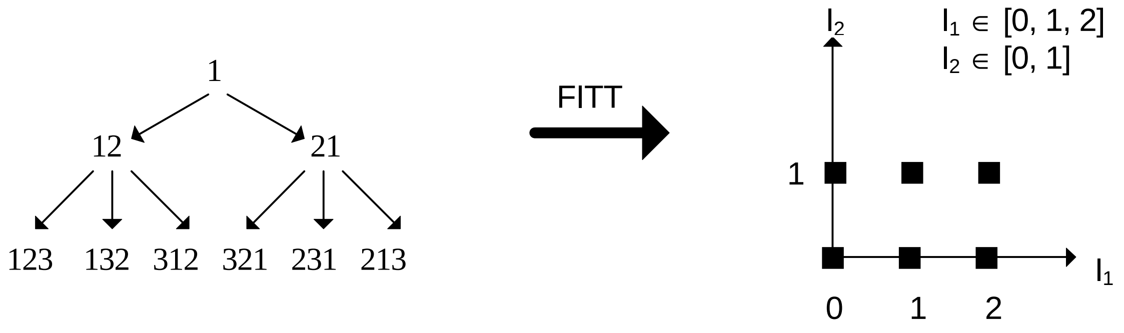

Fig. 5 : Fast Inversion Table Transform

Figure 5 on this page, graphically suggests how the FITT works. On the left there is one of the trees that generates all possible permutations of three elements and on the right there is the correspondent transformed vector space. Permutations of three elements in the example, are uniquely converted into vectors of two components that take values on the ranges : **[0, 1, 2]** and **[0, 1]** respectively**.** In other words a permutation space of ***n*** elements is transformed into a ***n - 1*** dimensional discrete linear space. This is another way of looking at the inversion table, as a convenient mapping between two spaces with different properties, more useful for optimization purposes.

For an interesting paper about the inversion table and permutation encoding, see also [Leino94], where an interesting application of the inversion table, such as inversion of programs as well as new algorithms, are also presented.

Algorithms for the Inversion Table

Algorithms to generate the inversion table from a permutation and back of complexity $O(n^2)$ are simple implementations where a linked list is used as the basic data structure and insertions and deletions are conveniently done. As pointed out in the previous chapters, given that the most interesting optimization problems deal with a large number of parameters, only algorithms $O(n \log n)$ or with better complexity performance can be efficiently used for this transformations. In this chapter the basic algorithms and their implementations are presented and described, for further details see [Knuth73]. Also, because the implementation of the FITT and its inverse, written in C++, turned out to be quite simple and easy to understand, instead of describing the algorithm using a mathematical formalism, supposedly more expressive, we decided that the programming language itself was more descriptive and simpler than any artificial notation. Therefore all the references to the algorithms will be directly done to the C++ implementation itself, listed in appendix of this report. The FITT and its inverse has been implemented in a C++ general class called **InvTab** which apart from its constructor and destructor has the two member functions **decodeInv** and **encodeInv** as the only public interface. As expected**, decodeInv** and **encodeInv** operate on permutations and vectors respectively. The general structure used to process the data is still a linked list and is built and initialized when the constructor of the class is invoked. In addition to the linked list, two arrays, **op,** with pointers at the elements in the list and **xs**, which is used as a counter, are utilized during the two transformations. Permutations are supposed to take integer values on the range **{ 1, 2 ... n }** and vectors on the range **{ 0, 1 ... n-1 }** with the last component always being zero. Notice that each single element of the list, called **item,** is a record of two values, where **digit** represent one element and **space** is the number of elements in front of it (on its left).

From permutations to vectors : $\{ a_1, a_2 \ldots a_n \} \rightarrow \{ b_1, b_2 \ldots b_n \}$

The implementation of a O(*n log n)* algorithm is much easier to understand in this case.
The operation required is to compute for each element at a given position in the permutation, the total number of smaller integers that precede it on its left. In order to make this operation efficient, a binary search tree is used to index all the elements $\mathbf{a_i}$, so that only *log n* levels must be updated. Each bit of $\mathbf{a_i}$, is accessed by an appropriate shift operation and the array **xs** is updated according to the value of that bit. The array **xs** is initialized with zeros at each of the *log n* iterations and **op** in the end will be pointing at the elements of type **item**, whose **space** field will be the index into the array of $\{ \mathbf{b_1, b_1 \ldots b_n} \}$ and **digit** will contain the appropriate value.

A left shift of one position is necessary in this case, because elements in the permutation take values on the range **{ 1, ... ,n }** at positions **{ 0, ... ,n-1 }**. The code is implemented in the procedure **encodeInv** listed in appendix of this report.

From vectors to permutations : $\{ b_1, b_2 \ldots b_n \} \rightarrow \{ a_1, a_2 \ldots a_n \}$

If a string (list) of element of type **item** such as $\alpha = [m_1, n_1], \ldots ,[m_n, n_n]$ and an *empty string* $\varepsilon = 0$ is given, we can define a binary composition $\otimes$ which takes two strings $([m, n]\alpha)$, $([m', n']\beta)$ where α , β are substrings without the first elements and creates a new string according to the rules :

$$\begin{tikzcd}
 & ([m, n](\alpha \otimes ([m' - m, n']\beta)) \quad \textit{if } m \leq m' \\
([m, n]\alpha) \otimes ([m', n']\beta) \arrow[ur] \arrow[dr] & \textbf{Eq. 2} \\
 & ([m', n']([m - m' - 1, n]\ \alpha) \otimes \beta) \quad \textit{if } m > m'
\end{tikzcd}$$

where $\varepsilon \otimes \alpha = \alpha \otimes \varepsilon = \alpha$ and $\otimes$ is associative : $\alpha \otimes (\beta \otimes \gamma) = (\alpha \otimes \beta) \otimes \gamma$

In this case it can be proved that :

$$[b_1, 1] \otimes [b_2, 2] \otimes \ldots \otimes [b_n, n] = [0, a_1]\ [0, a_2] \ldots [0, a_n] \qquad \textbf{Eq. 3}$$

or in words : the composition of a list of elements whose **space** is the inversion table value and **digit** goes from 1 to n, generates a list of elements whose **digit** field is the corresponding element of the permutation. The time to evaluate the above composition can also be shown to be *O(n log n)*. Notice that because $\otimes$ is a composition, therefore it is also associative, the expression on the left of Eq. 3, can be evaluated in any order. This is exactly what the private member function called **decode** does when invoked by the user called **decodeInv** with an inversion table as its input parameter. Notice that a *divide and conquer* recursive algorithm is used on the initial input list of the expression to be evaluated (left side of Eq. 3). In each iteration **i** the **op[i]** points to the result of the composition of the sub-strings that are being evaluated according to the rules established in Eq. 2. The end of a string is identified by an element with value 0 (*empty string*).

For example given the inversion table :

2 3 6 4 0 2 2 1 0

four iterations are needed to process an initial list of 9 elements :

$$[2, 1] \otimes [3, 2] \otimes [6, 3] \otimes [4, 4] \otimes [0, 5] \otimes [2, 6] \otimes [2, 7] \otimes [1, 8] \otimes [0, 9]$$

- 1st pass - after we evaluate adjacent pairs :

[2, 1] [1, 2] ⊗ [4, 4] [1, 3] ⊗ [0, 5] [2, 6] ⊗ [1, 8] [0, 7] ⊗ [0, 9]

- 2nd pass - after we evaluate adjacent sub-strings with 2 elements each :

[2, 1] [1, 2] [1, 4] [1, 3] ⊗ [0, 5] [1, 8] [0, 6] [0, 7] ⊗ [0, 9]

- 3rd pass - after we evaluate adjacent sub-strings with 4 elements each :

[0, 5] [1, 1] [0, 8] [0, 2] [0, 6] [0, 4] [0, 7] [0, 3] ⊗ [0, 9]

- 4th pass - will produce the resulting permutation :

[0, 5] [0, 9] [0, 1] [0, 8] [0, 2] [0, 6] [0, 4] [0, 7] [0, 3]

which is :

5 9 1 8 2 6 4 7 3

Algorithms that use balanced trees, for instance red-black trees, can also be used instead of the one presented here.
Moreover, if speed is a real concern, the recursive algorithm can also be rewritten in a iterative form, which also saves memory during the evaluation of the sub-expressions.

Performance Summary of the FITT Implementation

Some simple tests have been run on the implementation listed in this report and on a Digital Celebris Xl 6200 platform, a 200 Mhz Intel P6 system.
The code has been compiled with Microsoft Visual C++ ver. 4.2 and optimized for maximum speed. No other special custom optimizations have been selected and the reported results are averages over one thousand iterations on vectors with variable number of components of randomly generated values. Table 1 reports the execution times of **decodeInv** and **encodeInv** on permutations and vectors with different numbers of elements.

number of elements	decodeInv (average time per operation)	encodeInv (average time per operation)
500	0.94 ms	0.83 ms
1000	2.14 ms	1.47 ms
2000	4.23 ms	3.13 ms
4000	9.23 ms	6.81 ms
8000	24.00 ms	15.54 ms

Table 1 : FITT execution times

The Dynamic Hill Climbing example

In this chapter a technique first introduced by Yuret and De la Maza [YuMaza] and normally used to optimize objective functions of any kind in cartesian spaces where derivatives are not available or impossible to determine, will be applied to a difficult ordering problem in VLSI : the optimal placement in a plane of connected circuits or modules of various sizes. Usually one of the goals is that the total length of all the connections among the different modules be minimized, so that a given timing requirement can be met. Various algorithms using from simulated annealing techniques to genetic algorithms or evolution strategies have been conceived, carefully engineered and tuned to generate the best possible results. This example does not show that dynamic hill climbing is a better algorithm than others, its only purpose is to show that by using this approach we allow techniques that can only be used to optimize functions in cartesian spaces to also deal with ordering problems in a natural way. The quality of the final placement has been compared with the results obtained by running other two optimizers : a sophisticated tool such as TimberWolf ver. 7 that uses simulated annealing techniques and a genetic algorithms that has been implemented on a system developed here [Turr96] as an optimization research tool. The dynamic hill climber itself has a very straightforward implementation just to make the example possible, nevertheless the good results that came out from this experiment show that optimization methods that work on linear spaces can be extremely effective even when compared with problem specific highly engineered tools. As a final note, only placements of a limited number of modules (few hundreds) have been reported here.

If a real VLSI cell placement (tens of thousands of cells or more) has to be performed clustering techniques [see Turr96 pag. 14 - 21] should be added no matter which optimization algorithm is used. TimberWolf, in particular uses clustering by default, so the results provided by this report always reflect the time improvement that comes from that.

The Dynamic Hill Climbing algorithm

In this chapter dynamic hill climbing will only be briefly described, but readers interested in more details can look at [YuMaz94]. The code has been also changed to allow the algorithm to work in the discrete space generated by the FITT instead of a continuous one as originally conceived by the authors. For our purposes the algorithm can be easily described by a two nested loop structure. The outer loop which keeps exploring the search space as uniformly as possible, is described in the simple pseudo-code of [Fig. 6] by a loop that keep exploring the space around a given starting point *x*.

```
X = {};  // empty set
for (i = 0; i  < maxOptima; ++localOtimum)
{
        x = FarPoint(X);
        X = X ∪ LocalOptimize(f, x);
}
```

Fig. 6 : outer loop

where *f(x)* is the function to optimize taking a vector *x* as its input and *X* is the set of local optima already computed. *FarPoint* is a procedure which returns the new farthest point from all the ones already in the set *X*. Finally the procedure L*ocalOptimize* is another loop structure which given the objective function *f* and a point *x* return the best local optimal point, according to some rules.The loop keeps executing its body until *maxOptima* new points have been generated, implementing the idea of the so called *iterative deepening* by keeping exploring the space in increasing detail [see Fig.7.]

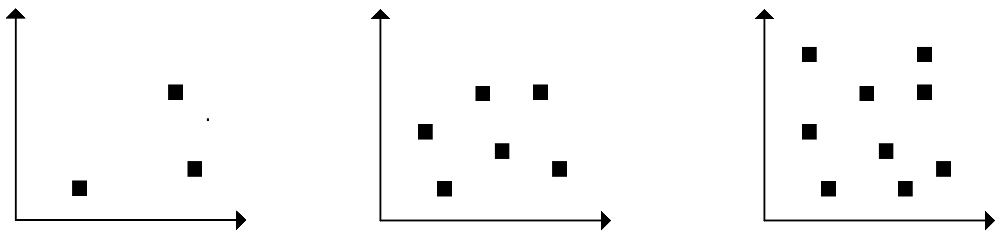

Fig. 7 : more iterations more local optima, better exploration

Let us concentrate now on the L*ocalOptimize*, the code of which is shown in Fig. 8. Essentially the idea is to have a starting point ***x*** and a probing vector, ***v***, whose length grows and shrinks depending on the value of the function at the new point : better points are rewarded, the vector length grows, worse point are penalized and ***v*** srinks. The coordinates of the best point so far are given by ***x* + *v***. Directions are randomly tried for a maximum of *maxIter* iterations with **|*v*|** vector length, until a better point is found. As we said before, if the new value is better than the previous one, the probing vector doubles in length and further regions of the space will be searched, otherwise the vector length is halved and regions closer to the local best optimum are sampled. To approximate and follow better the ridges in the mutidimensional search space, another vector ***u*** keeps the previous successful direction which made an improvement, is linearly combined with vector ***v*** [see Fig. 9] and the new promising direction, ***u* + *v*** , is tried out. The loop stops when the size of vector ***v*** decreases to a given minimum and the best solution is returned.

```
while( |v| >= threshold)
{
        iter = 0;
        while ( f(x + v) >= f(x)  &&  iter < maxIter)
        {
                v= randomVector(v);  ++iter;
        }
        if ( f(x + v) > f(v))
           v = v / 2;
        else  if( iter == 0)
            {
                x = x + v;  u = u + v;  v = 2 v;
            }
          else if( f(x + u + v) < f(v))
              {
                x = x + v + u;  u = u + v;  v = 2 u;
              }
            else { x = x + v;  u = v;  v = 2 v; }
}
```

Fig. 8 *:* LocalOptimize

The code in Fig. 8 is used to minimize the value of a given objective function *f(x)* until |*v*| gets smaller than a fixed threshold. In particular this algorithm is now integral part of the Genetic Workbench (GWB) [Turr96], a system developed at the Western Research Laboratory of Digital Equipment Corporation for experimenting optimization techniques on order-based problems. The results reported in the next chapter have been collected by running the GWB on circuits of various complexity and with increasing number of modules and connections.

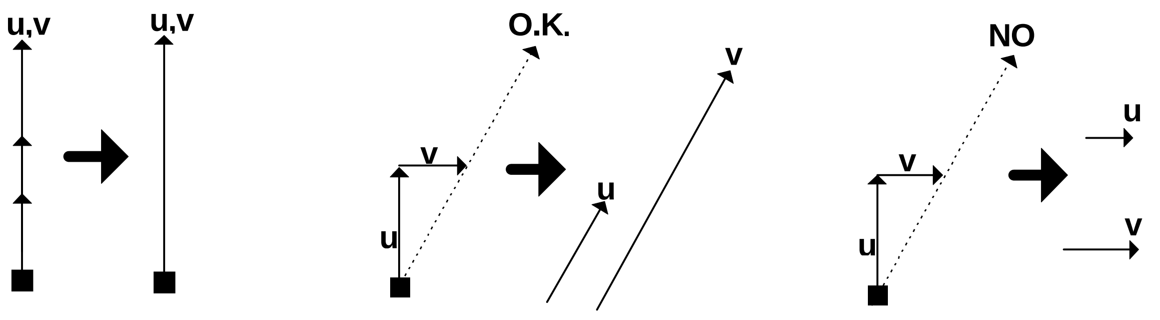

Fig. 9 : three cases illustrated (1 - no change in direction, 2 - successful try, 3 - failure)

The system has been extensively used on some problems and placement in particular so that the objective function and the utilities used to handle real circuits were already in place. Placements were described by a permutation of integers used to identify instances of cells to be placed on a predefined area on a plane. A routine was called to compute the actual coordinates of the cells by placing them in rows according to rules and constraints specified by the user. Every time the evaluation of the new placement was requested by the algorithm the inverse of the FITT would transform the vector back into a permutation and the placement routine just mentioned was called. The cost was approximated by computing the minimal spanning tree of the graph representing the connectivity of the circuit. So in this particular example the FITT is only used to allow a general algorithm like DHC to optimize an order-based problem such as the one described. For some other problems a better use of the transformed vectors is also possible if operations in the vector space are simpler than the ones in the permutation domain and the transformation preserves miniminality (or maximality) of the specific objective function.

Results

The results reported here are worst cases out of several runs of placements of real circuits with a relatively small number of cells that go from 28 to a maximum of 200. For this comparison two other candidates have been considered : the best genetic algorithm that has been developed for this specific problem under the Genetic Workbench [Turr96] and a commercial tool, TimberWolf ver. 7. Because TimberWolf was provided in executable form for a DecStation 5000, all the algorithms have been tested on the same platform. The quality of a placement has only been judged in terms of cost of the total length of all the connections. In order to do that consistently through all the examples an algorithm that computes the minimal spanning tree of the graph representing the circuit connections has been used. The execution time is also the only other parameter that has been used to compare different techniques. It should also be taken into account that TimberWolf was using a clustering algorithm in conjunction with a special simulated annealing schedule which dramatically improved the performance of the tool.

Legend : DHC = Dynamic Hill Climber
all values are in the form of : ***cost / time in seconds*** [read smaller better]

# cells (size)	**Genetic**	**DHC**	**TimberWolf**
28 cells	1624 / 400.0	1700 / 10.0	1813 / 54.2
96 cells	415 / 980.0	460 / 78.0	512 / 70.0
100 cells	552 / 1020.0	580 / 89.0	680 / 66.0
144 cells	980 / 2015.0	1118 / 280.0	1200 / 170.0
200 cells	1400 / 4000.0	1380 / 360.0	1480 / 210.0

Table 1 : comparison of DHC with other two specialized algorithms

Notice that the quality of the placements are always better for Dynamic Hill Climbing (DHC) than for TimberWolf and despite the lack of any kind of optimization in the case of DHC, even the execution times are not that different. For these examples the genetic algorithm produced the lowest cost placements, but the worst running times.

Appendix

```
//                              -----  Class definitions  -----
//                              -----  invTable.h  -----


#if !defined(TRANSFORM_DEF)
#define TRANSFORM_DEF

// Version described in D. Knuth's book (book 3 - exercise on permutations)

typedef struct item * pitem;
typedef int *  pGene;

struct item
{
        int    space;
        int    digit;
        pitem  next;
};

class InvTab
{
private:
   pitem   __fastcall decode(pitem, pitem);
   int       max;                                    // max vector length
   int        lim;                                           // 2^lim <= max < 2^(lim+1)
   pitem * op;
   int      * xs;
   pitem    pList;
public:
   InvTab(int = 8);
  ~InvTab();
   void __fastcall decodeInvTab(pGene, pGene);
   void __fastcall encodeInvTab(pGene, pGene);
};

//        -----  Inline implementations in the same translation unit (next page)  -----
```

```
inline pitem __fastcall InvTab::decode(pitem p1, pitem p2)  // recursive implementation
{
   pitem pT;

   if (!p1) return p2;
   else
          if (!p2) return p1;
          else
          {
                    if (p1->space <= p2->space)
                    {
                              p2->space = p2->space - p1->space;
                              p1->next = decode(p1->next, p2);
                    }
                    else
                    {
                              pT = p1;
                              p1 = p2;
                              p2 = pT;
                              p2->space -= (p1->space + 1);
                              p1->next = decode(p2, p1->next);
                    }
                    return p1;
          }
}

inline void __fastcall InvTab::decodeInvTab(pGene p, pGene q)
{
   pitem pT;
   int i, j, k, l;

   pT = op[0] = pList;
   for (i = 0; i < max;)                                // build and initialize the internal list
   {
      pT->space = p[i];
      pT->digit = ++i;
      pT->next = 0;
      op[i] = ++pT;
   }
   for (l = 1, k = 2; l < max; l *= 2, k *= 2)
      for (i = 0, j = l; j < max; i += k, j += k)
         op[i] = decode(op[i], op[j]);
   pT = op[0];
   for (i = 0; i < max; ++i, pT = pT->next)     // copy result
                    q[i] = pT->digit;
}
```

```
inline void __fastcall InvTab::encodeInvTab(pGene p, pGene q)
{
   pitem pT;
   int i, j, k, s, r;

   pT = op[0] = pList;
   for (i = 0; i < max;)                                        // initialize the internal list
   {
      pT->space = p[i];
      pT->digit = 0;
      op[++i] = ++pT;
   }
   for (k = lim; k >= 0; --k)
   {
      for (j = 0; j <= (max >> (k + 1)); ++j) xs[j] = 0;
      for (j = 0; j < max; ++j)
      {
         r = (op[j]->space >> k) % 2;
         s = op[j]->space >> (k + 1);
         if (r) ++xs[s]; else op[j]->digit += xs[s];
      }
   }
   for (i = 0; i < max; ++i)
   {
      q[op[i]->space - 1] = op[i]->digit;
   }
}

#endif
```

//   ----- more invTab.cpp on the next page -----

```
//                                   ----- invTable.ccp  -----

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "transform_1.h"

InvTab::InvTab(int s) : max(s)
{
   pitem pT1, pT2;

   if (s != 0)
   {
      op = new pitem [max + 1];                  // create list of items
                  pT1 = pList = new item [max + 1];
                  op[0] = pT1;
                  for (int i = 0; i < max; ++i)
                  {
                           pT2 = pT1 + 1;
                           pT1->next = pT2;
                           pT1 = pT2;
                  }
                  pT1->next = 0;
      xs = new int [max / 2 + 1];
      lim = int(log(double(max)) / log(2.0));
   }
   else
   {
      op = 0;
      xs = 0;
   }
}

InvTab::~InvTab()
{

   if (xs != 0) delete [] xs;
   if (op != 0)
   {
                  delete [] pList;
                  delete [] op;
   }
}
```

References

[CoTuk65] Cooley P.M. and J.W. Tukey, “An Algorithm for the Machine Computation of Complex Fourier Series” Mathematics of Computation. Vol. 19 (April 1965) 297-301

[Gold89] Goldberg David “Genetic Algorithms in Search Optimization & Machine Learning” Addison-Wesley Publishing Company, Inc. 1989 pp. 170 - 171

[Knuth73] Knuth D.E. “The Art of Computer Programming”, vol. 3 : Sorting and Searching. Addison Wesley

[Leino94] Leino K.R.M. “Computing Permutation Encodings”, Computer Science, California Institute of Technology, Pasadena, Caltech-CS-TR-94-12 ,1994

[Turr96] Turrini Silvio, “Optimizations and Placements with the Genetic WorkBench" DEC technical report 96/4.
[http://www.reserach.degital.com/techreports/abstracts/96.4/html]

[YuMaz94] Deniz Yuret and Michael De La Maza “Dynamic Hill Climbing”, article on AI Expert, March 1994, pag. 26-31.

WRL Research Reports

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’ **Norman P. Jouppi.** WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’ **Norman P. Jouppi.** WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’ **Norman P. Jouppi and Jeffrey Y. F. Tang.** WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’ **Norman P. Jouppi.** WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’ **Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.** WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’ **David W. Wall.** WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’ **Jeffrey Y.F. Tang and J. Leon Yang.** WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’ **Tracy Larrabee.** WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’ **Tracy Larrabee.** WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’ **Michael N. Nelson.** WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’ **Jeffrey C. Mogul.** WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’ **John S. Fitch.** WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’ **Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.** WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’ **Joel McCormack.** WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’ **J. Bradley Chen, Anita Borg, Norman P. Jouppi.** WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’ **Don Stark.** WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’ **David Boggs.** WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’ **Scott McFarling.** WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’ **Joel Bartlett.** WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’ **G. May Yip.** WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’ **William R. Hamburgen.** WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’ **David W. Wall.** WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’ **Jeffrey C. Mogul.** WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’ **Norman P. Jouppi.** WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’ **William R. Hamburgen, John S. Fitch.** WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’ **Jeffrey C. Mogul.** WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’ **David W. Wall.** WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’ **Russell Kao.** WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’ **Amitabh Srivastava and David W. Wall.** WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’ **Joel McCormack & Bob McNamara.** WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’ **Jeffrey C. Mogul.** WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’ **Norman P. Jouppi & Steven J.E. Wilton.** WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’ **Amitabh Srivastava.** WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’ **Steven J.E. Wilton and Norman P. Jouppi.** WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’ **David W. Wall.** WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’ **Alberto Makino, William R. Hamburgen, John S. Fitch.** WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’ **Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.** WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’ **Amitabh Srivastava, David W. Wall.** WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’ **Amitabh Srivastava, Alan Eustace.** WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’ **Keith I. Farkas, Norman P. Jouppi.** WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’ **Jeffrey C. Mogul.** WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo, Herve Touati.** WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’ **J. Bradley Chen, David W. Wall, and Anita Borg.** WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’ **Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.** WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’ **Keith I. Farkas, Norman P. Jouppi, and Paul Chow.** WRL Research Report 94/8, December 1994.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’ **Brian K. Reid and Christopher A. Kent.** WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’ **Christopher A. Kent.** WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’ **Joel McCormack.** WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?.’’ **John Ousterhout.** WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’ **Joel F. Bartlett.** WRL Technical Note TN-12, October 1989.

‘‘Characterization of Organic Illumination Systems.’’ **Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett.** WRL Technical Note TN-13, April 1989.

‘‘Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.’’ **Norman P. Jouppi.** WRL Technical Note TN-14, March 1990.

‘‘Limits of Instruction-Level Parallelism.’’ **David W. Wall.** WRL Technical Note TN-15, December 1990.

‘‘The Effect of Context Switches on Cache Performance.’’ **Jeffrey C. Mogul and Anita Borg.** WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’ **Aaron Goldberg and John Hennessy.** WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’ **David W. Wall.** WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion.’’ **Scott McFarling.** WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures.’’ **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach.’’ **John S. Fitch.** WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter.’’ **David Boggs.** WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS.’’ **Jeffrey C. Mogul.** WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package.’’ **Patrick D. Boyle.** WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics.’’ **Joel F. Bartlett.** WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0.’’ **Jeremy Dion & Louis Monier.** WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture.’’ **Amitabh Srivastava and David W. Wall.** WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors.’’ **Scott McFarling.** WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo and Herve Touati.** WRL Technical Note TN-37, June 1993.

‘‘Piecewise Linear Models for Rsim.’’ **Russell Kao, Mark Horowitz.** WRL Technical Note TN-40, December 1993.