4

Relative Position Indexing Approach

Daniel Lichtblau

Wolfram Research, Inc., 100 Trade Center Dr., Champaign, IL 61820, USA
danl@wolfram.com

Abstract. We introduce some standard types of combinatorial optimization problems, and indicate ways in which one might attack them using Differential Evolution. Our main focus will be on *indexing by relative position* (also known as *order based representation*); we will describe some related approaches as well. The types of problems we will consider, which are abstractions of ones from engineering, go by names such as *knapsack problems*, *set coverings*, *set partitioning*, and *permutation assignment*. These are historically significant types of problems, as they show up frequently, in various guises, in engineering and elsewhere. We will see that a modest amount of programming, coupled with a sound implementation of Differential Evolution optimization, can lead to good results within reasonable computation time. We will also show how Differential Evolution might be hybridized with other methods from combinatorial optimization, in order to obtain better results than might be found with the individual methods alone.

4.1 Introduction

The primary purpose of this chapter is to introduce a few standard types of combinatorial optimization problems, and indicate ways in which one might attack them using Differential Evolution. Our main focus will be on *indexing by relative position* (also known as *order based representation*); we will describe some related approaches as well. We will not delve much into why these might be regarded as "interesting" problems, as that would be a chapter– or, more likely, book– in itself. Suffice it to say that many problems one encounters in the combinatorial optimization literature have their origins in very real engineering problems, e.g. layout of hospital wings, electronic chip design, optimal task assignments, boolean logic optimization, routing, assembly line design, and so on. The types of problems we will consider, which are abstractions of the ones from engineering, go by names such as *knapsack problems*, *set coverings*, *set partitioning*, and *permutation assignment*. A secondary goal of this chapter will be to introduce a few ideas regarding hybridization of Differential Evolution with some other methods from optimization.

I will observe that, throughout this chapter at least, we regard Differential Evolution as a *soft* optimization tool. Methods we present are entirely heuristic in nature. We usually do not get guarantees of result quality; generally this must be assessed by independent means (say, comparison with other tactics such as random search or greedy algorithms, or a priori problem-specific knowledge). So we use the word *optimization* a bit loosely, and really what we usually mean is *improvement*. While this may seem to be bad from a theoretical point of view, it has advantages. For one, the field is relatively

G.C. Onwubolu and D. Davendra (Eds.): Differential Evolution, SCI 175, pp. 81–120.
springerlink.com

young and is amenable to an engineering mind set. One need not invent a new branch of mathematics in order to make progress. At the end of the chapter we will see a few directions that might, for the ambitious reader, be worthy of further pursuit.

Another caveat is that I make no claim as to Differential Evolution being the *best* method for the problems we will discuss. Nor do I claim that the approaches to be seen are the only, let alone best, ways to use it on these problems (if that were the case, this would be a very short book indeed). What I do claim is that Differential Evolution is quite a versatile tool, one that can be adapted to get reasonable results on a wide range of combinatorial optimization problems. Even more, this can be done using but a small amount of code. It is my hope to convey the utility of some of the methods I have used with success, and to give ideas of ways in which they might be further enhanced.

As I will illustrate the setup and solving attempts using *Mathematica* [13], I need to describe in brief how Differential Evolution is built into and accessed within that program. Now recall the general setup for this method. We have some number of vectors, or *chromosomes*, of continuous-valued *genes*. They *mate* according to a crossover probability, *mutate* by differences of distinct other pairs in the pool, and compete with a parent chromosome to see who moves to the next generation. All these are as described by Price and Storn, in their Dr. Dobbs Journal article from 1997 [10]. In particular, crossover and mutation parameters are as described therein. In *Mathematica* the relevant options go by the names of `CrossProbability`, `ScalingFactor`, and `SearchPoints`. Each variable corresponds to a gene on every chromosome. Using the terminology of the article, `CrossProbability` is the CR parameter, `SearchPoints` corresponds to NP (size of the population, that is, number of chromosome vectors), and `ScalingFactor` is F. Default values for these parameters are roughly as recommended in that article.

The function that invokes these is called `NMinimize`. It takes a `Method` option that can be set to `DifferentialEvolution`. It also takes a `MaxIterations` option that, for this method, corresponds to the number of generations. Do not be concerned if this terminology seems confusing. Examples to be shown presently will make it all clear.

One explicitly invokes Differential Evolution in *Mathematica* as follows.

NMinimize[objective, constraints,
variables, Method → {"DifferentialEvolution", methodoptions}, otheropts]

Here `methodoptions` might include setting to nondefault values any or all of the options indicated below. We will show usage of some of them as we present examples. Further details about these options may be found in the program documentation. All of which is available online; see, for example,
http://reference.wolfram.com/mathematica/ref/NMinimize.html

Here are the options one can use to control behavior of `NMinimize`. Note that throughout this chapter, code input is in **bold face**, and output, just below the input, is not.

Options[NMinimize`DifferentialEvolution]

$\{CrossProbability \to \frac{1}{2}$, InitialPoints → Automatic,
PenaltyFunction → Automatic, PostProcess → Automatic,

RandomSeed $\rightarrow 0$, ScalingFactor $\rightarrow \frac{3}{5}$,
SearchPoints $\rightarrow$ Automatic, Tolerance $\rightarrow 0.001\}$

There are some issues in internal implementation that need discussion to avoid later confusion. First is the question of how constraints are enforced. This is particularly important since we will often constrain variables to take on only integer values, and in specific ranges. For integrality enforcement there are (at least) two viable approaches. One is to allow variables to take on values in a continuum, but use penalty functions to push them toward integrality [2]. For example, one could add, for each variable x, a penalty term of the form $(x - \text{round}(x))^2$ (perhaps multiplied by some suitably large constant). `NMinimize` does not use this approach, but a user can choose to assist it to do so by explicitly using the `PenaltyFunction` method option. Another method, the one used by `NMinimize`, is to explicitly round all (real-valued) variables before evaluating in the objective function. Experience in the development of this function indicated this was typically the more successful approach.

This still does not address the topic of range enforcement. For example, say we are using variables in the range $\{1,\ldots,n\}$ to construct a permutation of n elements. If a value slips outside the range then we might have serious difficulties. For example, in low level programming languages such as C, having an out-of-bounds array reference can cause a program to crash. While this would not as likely happen in *Mathematica*, the effect would still be bad, for example a hang or garbage result due to processing of a meaningless symbolic expression. So it is important either that our code, or `NMinimize`, carefully enforce variable bounds. As it happens, the implementation does just this. If a variable is restricted to lie between a low and high bound (this is referred to as a *rectangular*, or *box*, constraint), then the `NMinimize` code will force it back inside the boundary. Here I should mention that this is really a detail of the implementation, and should not in general be relied upon by the user. I point it out so that the reader will not be mystified upon seeing code that blithely ignores the issue throughout the rest of this chapter. I also note that it is not hard to make alterations e.g. to an objective function, to preprocess input so that bound constraints are observed; in order to maximize simplicity of code, I did not do this.

I will make a final remark regarding use of *Mathematica* before proceeding to the material of this chapter. It is not expected that readers are already familiar with this program. A consequence is that some readers will find parts of the code we use to be less than obvious. This is to be expected any time one first encounters a new, complicated computer language. I will try to explain in words what the code is doing. Code will also be preceded by a concise description, in outline form, that retains the same order as the code itself and thus serves as a form of pseudocode. The code details are less important. Remember that the emphasis is on problem solving approaches using Differential Evolution in general; specifics of a particular language or implementation, while of independent interest, take a back seat to the Big Picture.

In the literature on combinatorial (and other) optimization via evolutionary means, one frequently runs across notions of *genotype* and *phenotype*. The former refers to the actual chromosome values. Recall the basic workings of Differential Evolution. One typically forms a new chromosome from mating its parent chromosome with a mutation of a

random second parent. Said mutation is in turn given as a difference of two other random chromosomes. These operations are all done at the genotype level. It is in translating the chromosome to a combinatorial object e.g. a permutation, that one encounters the phenotype. This refers, roughly, to the *expression* of the chromosome as something we can use for objective function evaluation. Said slightly differently, one *decodes* a genotype to obtain a phenotype. We want genotypes that are amenable to the mutation and mating operations of Differential Evolution, and phenotypes that will respond well to the genotype, in the sense of allowing for reasonable improvement of objective function. Discussion of these matters, with respect to the particulars of Differential Evolution, may be found in [11]. Early discussion of these issues, and methods for handling them, appear in [4] and [3].

4.2 Two Simple Examples

I like to start discussion of Differential Evolution in discrete optimization by presenting two fairly straightforward examples. They serve to get the reader acclimated to how we might set up simple problems, and also to how they look as input to *Mathematica*. These are relatively simple examples of discrete optimization, not involving combinatorial problems, and hence are good for easing into the main material of this chapter.

4.2.1 Pythagorean Triples

First we will search for Pythagorian triples. These, as one may recall from high school, are integer triples (x,y,z) such that $x^2+y^2=z^2$. So we wish to find integer triples that satisfy this equation. One way to set up such a problem is to form the square of the difference, $x^2+y^2-z^2$. We seek integer triples that make this vanish, and moreover this vanishing is a minimization condition (because we have a square). Note that this is to some extent arbitrary, and minimizing the absolute value rather than the square would suffice just as well for our purpose.

We constrain all variables to be between 5 and 25 inclusive. We also specify explicitly that the variables are integer valued. We will say a bit more about this in a moment.

```
NMinimize[{(x^2 + y^2 - z^2)^2, Element[{x, y, z}, Integers],
  5 <= x <= 25, 5 <= y <= 25, 5 <= z <= 25, x <= y}, {x, y, z}]
```

$$\{0.,\{x\to 7,y\to 24,z\to 25\}\}$$

We see that `NMinimize` is able to pick an appropriate method by default. Indeed, it uses `DifferentialEvolution` when variables are specified as discrete, that is, integer valued.

Now we show how to obtain different solutions by specifying that the random seed used by the `DifferentialEvolution` method change for each run. We will suppress warning messages (the algorithm mistakenly believes it is not converging). After all, we are only interested in the results; we can decide for ourselves quite easily if they work.

```
Quiet[
 Timing[
  Table[NMinimize[{(x^2 + y^2 - z^2)^2,
    Element[{x,y,z}, Integers], 5 ≤ x ≤ 25, 5 ≤ y ≤ 25, 5 ≤ z ≤ 25, x ≤ y},
    {x,y,z},
    Method → "DifferentialEvolution", RandomSeed → RandomInteger[1000]],
   {20}]]]
```

```
{17.1771, {{0., {x → 9, y → 12, z → 15}}, {0., {x → 15, y → 20, z → 25}},
   {0., {x → 6, y → 8, z → 10}}, {0., {x → 5, y → 12, z → 13}},
   {0., {x → 6, y → 8, z → 10}}, {0., {x → 7, y → 24, z → 25}},
   {0., {x → 15, y → 20, z → 25}}, {0., {x → 15, y → 20, z → 25}},
   {0., {x → 15, y → 20, z → 25}}, {0., {x → 8, y → 15, z → 17}},
   {0., {x → 5, y → 12, z → 13}}, {0., {x → 9, y → 12, z → 15}},
   {0., {x → 9, y → 12, z → 15}}, {0., {x → 5, y → 12, z → 13}},
   {0., {x → 6, y → 8, z → 10}}, {0., {x → 5, y → 12, z → 13}},
   {0., {x → 5, y → 12, z → 13}}, {0., {x → 5, y → 12, z → 13}},
   {0., {x → 5, y → 12, z → 13}}, {0., {x → 15, y → 20, z → 25}}}}
```

We observe that each of these is a valid Pythagorean triple (of course, there are several repeats). Recalling our objective function, any failure would appear as a false minimum, that is to say, a square integer strictly larger than zero.

4.2.1.1 A Coin Problem

We start with a basic coin problem. We are given 143,267 coins in pennies, nickels, dimes, and quarters, of total value $12563.29, and we are to determine how many coins might be of each type. There are several ways one might set up such a problem in `NMinimize`. We will try to minimize the sum of squares of differences between actual values and desired values of the two linear expressions implied by the information above. For our search space we will impose obvious range constraints on the various coin types. In order to obtain different results we will want to alter the seeding of the random number generator; this changes the random initial parameters used to seed the optimization code. That is why we specify the method with this option added. We will do 10 runs of this.

```
Timing[Table[
 {min, sol} = NMinimize[
  {(p + 5n + 10d + 25q - 1256329)^2 + (p + n + d + q - 143267)^2,
   {p,n,d,q} ∈ Integers, 0 ≤ p ≤ 1256329, 0 ≤ n ≤ 1256329/5,
   0 ≤ d ≤ 1256329/10, 0 ≤ q ≤ 1256329/25},
  {p,n,d,q}, MaxIterations → 1000,
  Method → {DifferentialEvolution, RandomSeed → Random[Integer, 1000]}],
 {10}]]
```

NMinimize::cvmit : Failed to converge to the
requested accuracy or precision within 1000 iterations.

$$\{229.634,\{\{0.,\{p\to 22554,n\to 70469,d\to 24978,q\to 25266\}\},$$
$$\{0.,\{p\to 4094,n\to 79778,d\to 42102,q\to 17293\}\},$$
$$\{0.,\{p\to 23139,n\to 64874,d\to 31502,q\to 23752\}\},$$
$$\{0.,\{p\to 26649,n\to 72620,d\to 15558,q\to 28440\}\},$$
$$\{0.,\{p\to 2914,n\to 76502,d\to 48358,q\to 15493\}\},$$
$$\{0.,\{p\to 9714,n\to 49778,d\to 73110,q\to 10665\}\},$$
$$\{0.,\{p\to 26019,n\to 26708,d\to 77782,q\to 12758\}\},$$
$$\{0.,\{p\to 58229,n\to 31772,d\to 19494,q\to 33772\}\},$$
$$\{0.,\{p\to 8609,n\to 70931,d\to 46674,q\to 17053\}\},$$
$$\{0.,\{p\to 35049,n\to 55160,d\to 25398,q\to 27660\}\}\}\}$$

We obtained valid solutions each time. Using only, say, 400 iterations we tend to get solutions about half the time and "near" solutions the other half (wherein either the number of coins and/or total value is off by a very small amount). Notice that this type of problem is one of constraint satisfaction. An advantage to such problems is that we can discern from the proposed solution whether it is valid; those are exactly the cases for which we get an object value of zero, with all constraints satisfied.

4.3 Maximal Determinants

In this section we illustrate a heuristic methods on certain extremal matrix problems of modest size. As motivation for looking at this particular problem, I remark that it is sometimes important to understand extremal behavior of random polynomials or matrices comprised of elements from a given set.

Below we apply knapsack-style optimization to study determinants of matrices of integers with all elements lying in the set {-1,0,1}. The problem is to minimize the determinant of such a matrix (since we can multiply any row by -1 and still satisfy the constraints, the smallest negative value corresponds to the largest positive value). We will make the simplifying assumption that all diagonal elements are 1. Strictly speaking this is not combinatorial optimization, but it is a close relative, and will help to get the reader acquainted with the programming commands we will be using in this chapter. Thus is also a good example with which to begin this chapter.

Our objective function is simply the determinant. We want it only to evaluate when the variables have been assigned numeric values. This is quite important because symbolic determinants are quite slow to compute. So we set up the function so that it is only defined when numeric values are plugged in.

```
detfunc[a:{{_?NumberQ..}..}]/;Length[a]==Length[First[a]]:=Det[a]
```

Our code will take a matrix dimension as argument, and also an optional argument specifying whether to print the constraints. We use that in a small problem to show the

constraints explicitly, so that the reader may check that we have set this up correctly. Before showing the actual code we first outline the process.

Outline of ***detMin***

1. Input: the dimension, and the parameter settings we will use for `NMinimize`.
2. Create a matrix of variables.
3. Create a set of constraints.
   All variables must be integers.
   All variables lie in the range $[-1,1]$.
   Variables corresponding to the diagonal elements are all set to 1.
4. Call `NMinimize` on the objective function, using the above constraints and taking program parameters from the argument list.
5. Return the optimum found by `NMinimize`, along with the matrix that gives this value.

Here is the actual program to do this.

```
detMin[n_, cp_, sp_, it_, printsetup_:False]:=Module[
 {mat, vars, problemlist, j, best},
 mat = Array[x, {n,n}];
 vars = Flatten[mat];
 problemlist =
  {detfunc[mat], Flatten[{Element[vars, Integers], Map[-1 ≤ # ≤ 1&, vars],
   Table[x[j, j] == 1, {j,n}]}]};
 If[printsetup, Print[problemlist[[2]]]];
 best = NMinimize[problemlist, vars, MaxIterations → it,
 Method → {DifferentialEvolution, CrossProbability → cp, SearchPoints → sp}];
 {best[[1]], mat/.best[[2]]}
 ]
```

Here is our result for three-by-three matrices. We also show the constraints for this small example.

```
Timing[{min, mat} = detMin[3, .1, 20, 20, True]]
```

$$\{(x[1,1]|x[1,2]|x[1,3]|x[2,1]|x[2,2]|x[2,3]|x[3,1]|x[3,2]|x[3,3]) \in \text{Integers},$$
$$-1 \le x[1,1] \le 1, -1 \le x[1,2] \le 1, -1 \le x[1,3] \le 1,$$
$$-1 \le x[2,1] \le 1, -1 \le x[2,2] \le 1, -1 \le x[2,3] \le 1,$$
$$-1 \le x[3,1] \le 1, -1 \le x[3,2] \le 1, -1 \le x[3,3] \le 1,$$
$$x[1,1] == 1, x[2,2] == 1, x[3,3] == 1\}$$

$$\{0.528033, \{-4., \{\{1,1,1\}, \{1,1,-1\}, \{1,-1,1\}\}\}\}$$

We obtain -4 as the minimum (can you do better?) We now try at dimension 7. We will use a larger search space and more iterations. Indeed, our option settings were determined by trial and error. Later we will say more about how this might systematically be done.

```
Timing[{min, mat} = detMin[7, .1, 80, 80]]
```

```
{54.6874, {-576., {{1, 1, -1, -1, 1, -1, 1}, {1, 1, -1, 1, -1, -1, -1},
   {1, -1, 1, 1, 1, -1, -1}, {-1, -1, -1, 1, 1, 1, 1}, {1, 1, -1, -1, 1, 1, -1},
   {1, 1, 1, 1, 1, 1, 1}, {1, -1, -1, -1, -1, 1, 1}}}}
```

Readers familiar with the *Hadamard bound* for absolute values of matrix determinants will recognize that the minimum must be no smaller than the ceiling of $-7^{\frac{7}{2}}$, or -907. (In brief, this bound is the product of the lengths of the rows of a matrix; for our family, the maxinal length of each row is $\sqrt{7}$. That this product maximizes the absolute value of the determinant can be observed from the fact that this absolute value is the volume of the rectangular prism formed by the row vectors of the matrix. This volume can be no larger than the product of their lengths; it achieves that value precisely when the rows are pairwise orthogonal.)

We can ask how good is the quality of our result. Here is one basis for comparison. A random search that took approximately twice as long as the code above found nothing smaller than -288. Offhand I do not know if -576 is the true minimum, though I suspect that it is.

It is interesting to see what happens when we try this with dimension increased to eight.

```
Timing[{min, mat} = detMin[8, 1/50, 100, 200]]
```

```
{222.618, {-4096., {{1, -1, 1, 1, 1, -1, -1, -1}, {-1, 1, -1, 1, 1, 1, -1, -1},
   {-1, 1, 1, 1, 1, -1, 1, 1}, {1, 1, -1, 1, -1, -1, -1, 1},
   {-1, -1, -1, -1, 1, -1, -1, 1}, {1, 1, 1, -1, 1, 1, -1, 1},
   {1, 1, -1, -1, 1, -1, 1, -1}, {1, -1, -1, 1, 1, 1, 1, 1}}}}
```

In this case we actually attained the Hadamard bound; one can check that the rows (and likewise the columns) are all pairwise orthogonal, as must be the case in order to attain the Hadamard bound. Indeed, when the dimension is a power of two, one can always attain this bound. The motivated reader might try to work out a recursive (or otherwise) construction that gives such pairwise orthogonal sets.

4.4 Partitioning a Set

The last sections were a warmup to the main focus of this chapter. We introduced a bit of *Mathematica* coding, and in particular use of Differential Evolution, in the contect of discrete optimization. We now get serious in discussing combinatorial optimization problems and techniques.

We start with the *Set Partitioning Problem*. We will illustrate this with an old example from computational folklore: we are to partition the integers from 1 to 100 into two sets of 50, such that the sums of the square roots in each set are as close to equal as possible.

There are various ways to set this up as a problem for `NMinimize`. We will show two of them. First we will utilize a simple way of choosing 50 elements from a set of 100. We will use 100 real values, all between 0 and 1. (Note that we are using continuous variables even though the problem itself involves a discrete set.) We take their *relative positions* as defining a permutation of the integers from 1 to 100. A variant of this approach to decoding permutations is described in [4, 3].

In more detail: their sorted ordering (obtained, in our code, from the *Mathematica* `Ordering` function) determines which is to be regarded as first, which as second, and so on. As this might be confusing, we illustrate the idea on a smaller set of six values. We begin with our range of integers from 1 to 6.

smallset = Range[6]

$\{1,2,3,4,5,6\}$

Now suppose we also have a set of six real values between 0 and 1.

vals = RandomReal[1,{6}]

$\{0.131973, 0.80331, 0.28323, 0.694475, 0.677346, 0.255748\}$

We use this second set of values to split `smallset` into two subsets of three, simply by taking as one such subset the elements with positions corresponding to those of the three smallest member of `vals`. The complementary subset would therefore be the elements with positions corresponding to those of the three largest members of `vals`. One can readily see (and code below will confirm) that the three smallest elements of `vals`, in order of increasing size, are the first, sixth, and third elements.

Ordering[vals]

$\{1,6,3,5,4,2\}$

We split this into the positions of the three smallest, and those of the three largest, as below.

{smallindices, largeindices} = {Take[#,3], Drop[#,3]}&[Ordering[vals]]

$\{\{1,6,3\},\{5,4,2\}\}$

We now split smallset according to these two sets of indices. Because it is simply the values one through six, the subsets are identical to their positions.

{s1, s2} = Map[smallset[[#]]&, {smallindices, largeindices}]

$\{\{1,6,3\},\{5,4,2\}\}$

The same idea applies to splitting any set of an even number of elements (small modifications could handle an odd number, or a split into subsets of unequal lengths).

With this at hand we are now ready to try our first method for attacking this problem.

4.4.1 Set Partitioning via Relative Position Indexing

Here is the code we actually use to split our 100 integers into two sets of indices.

*Outline of **splitRange***

1. Input: a vector of real numbers, of even length.
2. Return the positions of the smaller half of elements, followed by those of the larger half.

```
splitRange[vec_]:=With[
  {newvec = Ordering[vec], halflen = Floor[Length[vec]/2]},
  {Take[newvec, halflen], Drop[newvec, halflen]}]
```

Just to see that it works as advertised, we use it to replicate the result from our small example above.

```
splitRange[vals]
```

$$\{\{1,6,3\},\{5,4,2\}\}$$

Once we have a way to associate a pair of subsets to a given set of 100 values in the range from 0 to 1, we form our objective function. A convenient choice is simply an absolute value of a difference; this is often the case in optimization problems. We remark that squares of differences are also commonly used, particularly when the optimization method requires differentiability with respect to all program variables. This is not an issue for Differential Evolution, as it is a derivative-free optimization algorithm.

Here is an outline of the objective function, followed by the actual code.

*Outline of **spfun***

1. Input: a vector of real numbers, of even length.
2. Use splitRange to find positions of the smaller half of elements, and the positions of the larger half.
3. Sum the square roots of the first set of positions, and likewise sum the square roots of the second set.
4. Return the absolute value of the difference of those two sums.

```
spfun[vec : {__Real}]:=
 With[{vals = splitRange[vec]},
  Abs[(Apply[Plus, Sqrt[N[First[vals]]]] - Apply[Plus, Sqrt[N[Last[vals]]]])]]
```

It may be a bit difficult to see what this does, so we illustrate again on our small example. Supposing we have split `smallset` into two subsets as above, what is the objective function? Well, what we do is take the first, sixth, and third elements, add their square roots, and do likewise with the fifth, fourth, and second elements. We subtract one of these sums from the other and take the absolute value of this difference. For speed we do all of this in machine precision arithmetic. In exact form it would be:

```
sqrts = Sqrt[splitRange[vals]]
```

$$\left\{\left\{1, \sqrt{6}, \sqrt{3}\right\}, \left\{\sqrt{5}, 2, \sqrt{2}\right\}\right\}$$

```
sums = Total[sqrts, {2}]
```

$$\left\{1+\sqrt{3}+\sqrt{6}, 2+\sqrt{2}+\sqrt{5}\right\}$$

```
sumdifference = Apply[Subtract, sums]
```

$$-1-\sqrt{2}+\sqrt{3}-\sqrt{5}+\sqrt{6}$$

```
abssummdiffs = Abs[sumdifference]
```

$$1+\sqrt{2}-\sqrt{3}+\sqrt{5}-\sqrt{6}$$

```
approxabs = N[abssummdiffs]
```

0.468741

As a check of consistency, observe that this is just what we get from evaluating our objective function on `vals`.

```
spfun[vals]
```

0.468741

We now put these components together into a function that provides our set partition.

Outline of ***getHalfSet***

1. Input: An even integer n, and options to pass along to `NMinimize`.
2. Create a list of variables, vars, of length n.
3. Set up initial ranges that the variables all lie between 0 and 1 (these are not hard constraints but just tell NMinimize where to take random initial values).

4. Call `NMinimize`, passing it `obfun[vars]` as objective function.
5. Return the minimum value found, and the two complementary subsets of the original integer set $\{1,\ldots,n\}$ that give rise to this value.

```
getHalfSet[n_, opts___Rule]:=Module[{vars, xx, ranges, nmin, vals},
 vars = Array[xx, n];
 ranges = Map[{#, 0, 1}&, vars];
 {nmin, vals} = NMinimize[spfun[vars], ranges, opts];
 {nmin, Map[Sort, splitRange[vars/.vals]]}]
```

As in previous examples, we explicitly set the method so that we can more readily pass it nondefault method-specific options. Finally, we set this to run many iterations with a lot of search points. Also we turn off post-processing. Why do we care about this? Well, observe that our variables are not explicitly integer valued. We are in effect fooling `NMinimize` into doing a discrete (and in fact combinatorial) optimization problem, without explicit use of discrete variables. Hence default heuristics are likely to conclude that we should attempt a "local" optimization from the final configuration produced by the differential evolution code. This will almost always be unproductive, and can take considerable time. So we explicitly disallow it. Indeed, if we have the computation time to spend, we are better off increasing our number of generations, or the size of each generation, or both.

```
Timing[{min, {s1, s2}} =
 getHalfSet[100, MaxIterations → 10000,
  Method → {DifferentialEvolution, CrossProbability → .8,
   SearchPoints → 100, PostProcess → False}]]
```

```
{2134.42, {2.006223098760529*^-7,
  {{1, 2, 4, 6, 7, 11, 13, 15, 16, 17, 19, 21, 23, 25, 26, 27, 31, 34,
    37, 41, 43, 44, 45, 47, 50, 51, 52, 54, 56, 66, 67, 69, 72, 73,
    75, 77, 78, 79, 80, 86, 87, 88, 89, 90, 91, 93, 96, 97, 98, 100},
   {3, 5, 8, 9, 10, 12, 14, 18, 20, 22, 24, 28, 29, 30, 32, 33, 35, 36,
    38, 39, 40, 42, 46, 48, 49, 53, 55, 57, 58, 59, 60, 61, 62, 63, 64,
    65, 68, 70, 71, 74, 76, 81, 82, 83, 84, 85, 92, 94, 95, 99}}}}
```

We obtain a fairly small value for our objective function. I do not know if this in fact the global minimum, and the interested reader might wish to take up this problem with an eye toward obtaining a better result.

A reasonable question to ask is how would one know, or even suspect, where to set the `CrossProbability` parameter? A method I find useful is to do "tuning runs". What this means is we do several runs with a relatively small set of search points and a fairly low bound on the number of generations (the `MaxIterations` option setting, in `NMinimize`). Once we have a feel for which values seem to be giving better results, we use them in the actual run with options settings at their full values. Suffice it to say

that this approach is far from scientific. About the best one can say is that, while it is not obviously fantastic, it is also not obviously bad. Note that this sort of situation happens often in engineering, and that is why one can make nice incremental improvements to a technology such as optimization.

4.4.2 Set Partitioning via Knapsack Approach

Another approach to this problem is as follows. We take the full set and pick 100 corresponding random integer values that are either 0 or 1. An element in the set is put into one or the other subset according to the value of the bit corresponding to that element. For this to give an even split we also must impose a constraint that the size of each subset is half the total size. To get an idea of what these constraints are, we show again on our small example of size six.

vars = Array[x,6];
ranges = Map[(0<=#<=1)&, vars];
Join[ranges, {Element[vars, Integers], Apply[Plus, vars] == 3}]

$$\{0 \le x[1] \le 1, 0 \le x[2] \le 1, 0 \le x[3] \le 1, 0 \le x[4] \le 1,$$
$$0 \le x[5] \le 1, 0 \le x[6] \le 1, (x[1]|x[2]|x[3]|x[4]|x[5]|x[6]) \in \text{Integers},$$
$$x[1]+x[2]+x[3]+x[4]+x[5]+x[6] == 3\}$$

We are now ready to define our new objective function.

Outline of ***spfun2***

1. Input: a vector of integers, of even length n. All entries are 0 or 1.
2. Convert every 0 to -1.
3. Form a list of square roots of the integers in $\{1,\ldots,n\}$.
4. Multiply, componentwise, with the list of ones and negative ones.
5. Return the absolute value of the sum from step (4).

spfun2[vec : {_Integer}]:=Abs[(2 * vec − 1).Sqrt[N[Range[Length[vec]]]]]

Again we use our small example. What would our objective function be if the vector has ones in the first two and last places, and zeros in the middle three? First we find the exact value.

exactval = Abs[Total[Sqrt[smallset[[{1,6,3}]]]] − Total[Sqrt[smallset[[{5,4,2}]]]]]

$1+\sqrt{2}-\sqrt{3}+\sqrt{5}-\sqrt{6}$

***N*[exactval]**

0.468741

We see that, as expected, this agrees with our objective function.

```
spfun2[{1,0,1,0,0,1}]
```

0.468741

With this knowledge it is now reasonably straightforward to write the code that will perform our optimization. We create a set of variables, one for each element in the set. We constrain the variables to take on values that are either 0 or 1, and such that the sum is exactly half the cardinality of the set (that is, 100/2, or 50, in the example of interest to us). Since we force variables to be integer valued, `NMinimize` will automatically use `DifferentialEvolution` for its method. Again, we might still wish to explicitly request it so that we can set option to nondefault values.

Outline of ***getHalfSet2***

1. Input: An even integer n, and options to pass along to `NMinimize`.
2. Create a list of variables, vars, of length n.
3. Set up constraints.
   All variables lie between 0 and 1.
   All variables are integers.
   Their total is $\frac{n}{2}$.
4. Call `NMinimize`, passing it `spfun2[vars]` as objective function, along with the constraints and the option settings that were input.
5. Return the minimum value found, and the two complementary subsets of the original integer set $\{1,\ldots,n\}$ that give rise to this value.

```
getHalfSet2[n_, opts___]:=Module[
 {vars,x,nmin,vals,ranges,s1},
 vars = Array[x,n];
 ranges = Map[(0 ≤ # ≤ 1)&,vars];
 {nmin,vals} =
  NMinimize[{spfun2[vars],
  Join[ranges,{Element[vars,Integers],Total[vars] == n/2}]},vars,opts];
 s1 = Select[Inner[Times,Range[n],(vars/.vals),List],# ≠ 0&];
 {nmin,{s1,Complement[Range[n],s1]}}]

Timing[
 {min,{s1,s2}} = getHalfSet2[100,MaxIterations → 1000,Method →
  {DifferentialEvolution,CrossProbability → .8,SearchPoints → 100}]]
```

```
{1732.97,{0.000251303,
   {1,4,5,7,12,13,14,15,16,19,20,22,23,31,32,36,37,38,41,42,
    43,44,45,46,47,49,50,51,52,55,59,60,62,65,66,71,73,78,
    79,83,84,87,88,89,90,91,94,97,99,100},
```

$$\{2,3,6,8,9,10,11,17,18,21,24,25,26,27,28,29,30,33,34,35,$$
$$39,40,48,53,54,56,57,58,61,63,64,67,68,69,70,72,74,75,$$
$$76,77,80,81,82,85,86,92,93,95,96,98\}\}\}$$

One unfamiliar with the subject might well ask what this has to do with knapsacks. The gist is as follows. A *Knapsack Problem* involves taking, or not taking, an element from a given set, and attempting to optimize some condition that is a function of those elements taken. There is a large body of literature devoted to such problems, as they subsume the *Integer Linear Programming Problem* (in short, linear program, but with variables constrained to be integer valued). It is a pleasant quality of Differential Evolution that it can be adapted to such problems.

4.4.3 Discussion of the Two Methods

The second method we showed is a classical approach in integer linear programming. One uses a set of variables constrained to be either 0 or 1 (that is, *binary* variables). We constrain their sum so that we achieve a particular goal, in this case it is that exactly half be put into one of the two subsets. While not quite a relative position indexing method, it is similar in that positions of zeros or ones determine which of two complementary subsets receives elements of the parent set.

The first method, which seemed to work better for Differential Evolution (at least with parameter settings we utilized) is less common. It is a bit mysterious, in that we use the ordering of an ensemble of reals to determine placement of individual elements of a set. This implies a certain *nonlocality* in that a change to one value can have a big effect on the interpretation of other entries. This is because it is their overall sorted ordering, and not individual values, that gets used by the objective function. Though it is not obvious that this would be useful, we saw in this example that we can get a reasonably good result.

4.5 Minimal Covering of a Set by Subsets

The problem below was once posed in the Usenet news group comp.soft-sys.math.mathematica. It is an archetypical example of the classical *subset covering problem*. In this example we are given a set of sets, each containing integers between 1 and 64. Their union is the set of all integers in that range, and we want to find a set of 12 subsets that covers that entire range. In general we would want to find a set of subsets of minimal cardinality; this is an instance where we know in advance that that cardinality is 12.

$$\mathbf{subsets} = \{\{1,2,4,8,16,32,64\},\{2,1,3,7,15,31,63\},\{3,4,2,6,14,30,62\},$$
$$\{4,3,1,5,13,29,61\},\{5,6,8,4,12,28,60\},\{6,5,7,3,11,27,59\},$$
$$\{7,8,6,2,10,26,58\},\{8,7,5,1,9,25,57\},\{9,10,12,16,8,24,56\},$$
$$\{10,9,11,15,7,23,55\},\{11,12,10,14,6,22,54\},\{12,11,9,13,5,21,53\},$$
$$\{13,14,16,12,4,20,52\},\{14,13,15,11,3,19,51\},\{15,16,14,10,2,18,50\},$$
$$\{16,15,13,9,1,17,49\},\{17,18,20,24,32,16,48\},\{18,17,19,23,31,15,47\},$$
$$\{19,20,18,22,30,14,46\},\{20,19,17,21,29,13,45\},\{21,22,24,20,28,12,44\},$$

```
{22,21,23,19,27,11,43},{23,24,22,18,26,10,42},{24,23,21,17,25,9,41},
{25,26,28,32,24,8,40},{26,25,27,31,23,7,39},{27,28,26,30,22,6,38},
{28,27,25,29,21,5,37},{29,30,32,28,20,4,36},{30,29,31,27,19,3,35},
{31,32,30,26,18,2,34},{32,31,29,25,17,1,33},{33,34,36,40,48,64,32},
{34,33,35,39,47,63,31},{35,36,34,38,46,62,30},{36,35,33,37,45,61,29},
{37,38,40,36,44,60,28},{38,37,39,35,43,59,27},{39,40,38,34,42,58,26},
{40,39,37,33,41,57,25},{41,42,44,48,40,56,24},{42,41,43,47,39,55,23},
{43,44,42,46,38,54,22},{44,43,41,45,37,53,21},{45,46,48,44,36,52,20},
{46,45,47,43,35,51,19},{47,48,46,42,34,50,18},{48,47,45,41,33,49,17},
{49,50,52,56,64,48,16},{50,49,51,55,63,47,15},{51,52,50,54,62,46,14},
{52,51,49,53,61,45,13},{53,54,56,52,60,44,12},{54,53,55,51,59,43,11},
{55,56,54,50,58,42,10},{56,55,53,49,57,41,9},{57,58,60,64,56,40,8},
{58,57,59,63,55,39,7},{59,60,58,62,54,38,6},{60,59,57,61,53,37,5},
{61,62,64,60,52,36,4},{62,61,63,59,51,35,3},{63,64,62,58,50,34,2},
{64,63,61,57,49,33,1}};
```

We do a brief check that the union of the subset elements is indeed the set of integers from 1 through 64.

```
Union[Flatten[subsets]] == Range[64]
```

True

4.5.1 An Ad Hoc Approach to Subset Covering

We will set up our objective function as follows. We represent a set of 12 subsets of this master set by a set of 12 integers in the range from 1 to the number of subsets (which in this example is, coincidently, also 64). This set is allowed to contain repetitions. Our objective function to minimize will be based on how many elements from 1 through 64 are "covered". Specifically it will be 2 raised to the #(elements not covered) power. The code below does this.

Outline of ***scfun***

1. Input: a vector V of integers, a set S of subsets, and an integer n to denote the range of integers $\{1,\ldots,n\}$.
2. Compute U, the union of elements contained in the subsets S_j, for all $j \in V$.
3. Calculate c, the cardinality of the complement of our initial range by U.
   More succinctly this is $|\{1,\ldots,n\} - U|$, where subtraction is taken to mean *set complement*, and $|S|$ denotes the cardinality of S.
4. Return 2^c.

```
scfun[n : {__Integer}, set_, mx_Integer]:=
 2^Length[Complement[Range[mx], Union[Flatten[set[[n]]]]]]
```

This may be a bit elusive. We will examine its behavior on a specific set of subsets. Suppose we take the first 12 of our subsets.

first12 = Take[subsets, 12]

{{1,2,4,8,16,32,64},{2,1,3,7,15,31,63},{3,4,2,6,14,30,62},
{4,3,1,5,13,29,61},{5,6,8,4,12,28,60},{6,5,7,3,11,27,59},
{7,8,6,2,10,26,58},{8,7,5,1,9,25,57},{9,10,12,16,8,24,56},
{10,9,11,15,7,23,55},{11,12,10,14,6,22,54},{12,11,9,13,5,21,53}}

Their union is

elementsinfirst12 = Union[Flatten[first12]]

{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,21,22,23,24,25,26,27,28,
29,30,31,32,53,54,55,56,57,58,59,60,61,62,63,64}

Our objective function for this set of subsets raises 2 to the power that is the cardinality of the set of integers 1 through 64 complemented by this set. So how many elements does this union miss?

missed = Complement[Range[64], elementsinfirst12]

{17,18,19,20,33,34,35,36,37,38,39,40,41,42,43,44,
45,46,47,48,49,50,51,52}

Length[missed]

24

2^Length[missed]

16777216

Does this agree with the function we defined above? Indeed it does.

scfun[Range[12], subsets, 64]

16777216

We now give outline and code to find a set of spanning subsets.

Outline of ***spanningSets***

1. Input: a set S of m subsets, an integer k specifying how many we are to use for our cover, and option values to pass to `NMinimize`. We assume the union of all subsets covers some range $\{1,\ldots,n\}$.
2. Create a vector of k variables.

3. Set up constraints.
   All variables are between 1 and m.
   All variables are integer valued.
4. Call `NMinimize`, using the constraints and scfun as defined above, along with option settings.
5. Return the minimal value (which we want to be 1, in order that there be full coverage), and the list of positions denoting which subsets we used in the cover.

```
spanningSets[set_, nsets_, iter_, sp_, cp_]:=Module[
 {vars, rnges, max = Length[set], nmin, vals},
 vars = Array[xx, nsets];
 rnges = Map[(1 ≤ # ≤ max)&, vars];
 {nmin, vals} = NMinimize[
  {scfun[vars, set, max], Append[rnges, Element[vars, Integers]]},
  vars, MaxIterations → iter,
  Method → {DifferentialEvolution, SearchPoints → sp, CrossProbability → cp}];
 vals = Union[vars/.vals];
 {nmin, vals}]
```

In small tuning runs I found that a fairly high crossover probability setting seemed to work well.

```
Timing[{min, sets} = spanningSets[subsets, 12, 700, 200, .94]]
```

```
{365.099, {1., {1, 7, 14, 21, 24, 28, 34, 35, 47, 52, 54, 57}}}
```

```
Length[Union[Flatten[subsets[[sets]]]]]
```

```
64
```

While this is not lightning fast, we do obtain a good result in a few minutes of run time.

We note that while this was not coded explicitly to use relative position indexing, it could have been. That is, we could have used vectors of 64 values between 0 and 1, and taken the positions of the smallest 12 to give 12 members of subsets. The interested reader may wish to code this variant.

4.5.2 Subset Covering via Knapsack Formulation

Another method is to cast this as a standard knapsack problem. First we transform each of our set of subsets into a *bit vector* representation. In this form each subset is represented by a positional list of zeros and ones. In effect we are translating from a *sparse* to a *dense* representation.

Outline of ***densevec***

1. Input: a set S of integers and a length n. It is assumed that the members of S all lie in $\{1,\ldots,n\}$.
2. Create a vector V of length n. Initialize all elements to be 0.
3. Loop: For each $j \in S$, set the jth element of V to be 1.
4. Return V.

```
densevec[spvec_, len_]:=Module[
  {vec = Table[0, {len}]},
 Do[vec[[spvec[[j]]]] = 1, {j, Length[spvec]}];
 vec]
```

We now apply this function to each member of our set of subsets, that is, make a dense representation of each subset.

```
mat = Map[densevec[#, 64]&, subsets];
```

It might not be obvious what we have done, so we illustrate using the fourth of our 64 matrix rows.

```
mat[[4]]
```

```
{1,0,1,1,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0}
```

We have ones at positions that correspond to the elements contained in our fourth subset, and zeros elsewhere. Specifically, the ones are at the positions shown below.

```
Flatten[Position[mat[[4]], 1]]
```

```
{1,3,4,5,13,29,61}
```

But this is, up to ordering, exactly the elements in the fourth subset. That is, we pass a basic consistency check.

```
Sort[subsets[[4]]]
```

```
{1,3,4,5,13,29,61}
```

As in our last knapsack problem, we again work with binary variables and minimize their sum, subject to certain constraints. We use a binary variables to represent each subset. A one means we use that subset in our set cover, and a zero means we do not.

Let us consider the vector of those zeros and ones. Now our requirement is that we fully cover the superset, that is, the range of integers from 1 to 64.

How might we impose this? Well, let us take a look at the dot product of such a vector with the matrix of bit vectors that we already formed. Again we use as an example the first 12 subsets, so our vector representing this has ones in the first 12 slots, and zeros in the remaining 64-12=52 slots.

first12vec = Join[ConstantArray[1, 12], ConstantArray[0, 52]]

$\{1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,$
$0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}$

What does the dot product of this vector with our matrix of bit vectors represent? Well, let's consider the meaning of this matrix for a moment. A one in row j, column k means that subset j contains element k. One then realizes that the dot product gives us the following information. The kth element in the result will be a nonnegative number (possibly zero), and represent the number of times that k appears in the union of subsets represented by `first12vec`. So the condition we will need to impose in our optimization is that the dot product of this vector with our matrix of bitvectors has all positive entries. Notice that `first12vec` fails to satisfy this condition.

first12vec.mat

$\{4,4,4,4,5,5,5,5,4,4,4,4,2,2,2,2,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,$
$0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1\}$

Outline of ***spanningSets2***

1. Input: a set S of m subsets (each now represented as a bit vector), and option values to pass to `NMinimize`. We assume the union of all subsets covers some range $\{1,\ldots,n\}$.
2. Create a vector of m variables, `vars`.
3. Set up constraints.
   All variables lie between 0 and 1.
   All variables are integer valued.
   The union of subsets corresponding to variables with value of 1 covers the full range $\{1,\ldots,n\}$. This is done by checking that each elements of S.vars is greater or equal to 1.
4. Call `NMinimize` to minimize the sum of vars, subject to the above constraints, using the input option settings.
5. Return the minimal value and the list of positions denoting which subsets we used in the cover.

```
spanningSets2[set_, iter_, sp_, seed_, cp_:.5]:=Module[
  {vars, rnges, max = Length[set], nmin, vals},
  vars = Array[xx, max];
  rnges = Map[(0 ≤ # ≤ 1)&, vars];
  {nmin, vals} =
  NMinimize[{Apply[Plus, vars], Join[rnges, {Element[vars, Integers]},
    Thread[vars.set ≥ Table[1, {max}]]]}, vars, MaxIterations → iter,
    Method → {DifferentialEvolution, CrossProbability → cp,
      SearchPoints → sp, RandomSeed → seed}];
  vals = vars/.vals;
  {nmin, vals}]

Timing[{min, sets} = spanningSets2[mat, 2000, 100, 0, .9]]

{1930.4Second, {12., {0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
   0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}}}
```

We have again obtained a result that uses 12 subsets. We check that it covers the entire range.

```
Total[Map[Min[#, 1]&, sets.mat]]

64
```

We see that this method was much slower. Experience indicates that it needs a lot of iterations and careful setting of the CrossProbability option. So at present `NMinimize` has difficulties with this formulation. All the same it is encouraging to realize that one may readily set this up as a standard knapsack problem, and still hope to solve it using Differential Evolution. Moreover, as the alert reader may have observed, we actually had an added benefit from using this method: nowhere did we need to assume that minimal coverings require 12 subsets.

4.6 An Assignment Problem

Our next example is a benchmark from the literature of discrete optimization. We are given two square matrices. We want a permutation that, when applied to the rows and columns of the second matrix, multiplied element-wise with corresponding elements of the first, and all elements summed, gives a minimum value. The matrices we use have 25 rows. This particular example is known as the NUG25 problem. It is an example of a *Quadratic Assignment Problem* (QAP). The optimal result is known and was verified by a large parallel computation. We mention that the methods of handling this problem can, with minor modification, be applied to related problems that require the selecting of a permutation (for example, the traveling salesman problem).

```
mat1 = {{0,1,2,3,4,1,2,3,4,5,2,3,4,5,6,3,4,5,6,7,4,5,6,7,8},
  {1,0,1,2,3,2,1,2,3,4,3,2,3,4,5,4,3,4,5,6,5,4,5,6,7},
  {2,1,0,1,2,3,2,1,2,3,4,3,2,3,4,5,4,3,4,5,6,5,4,5,6},
  {3,2,1,0,1,4,3,2,1,2,5,4,3,2,3,6,5,4,3,4,7,6,5,4,5},
  {4,3,2,1,0,5,4,3,2,1,6,5,4,3,2,7,6,5,4,3,8,7,6,5,4},
  {1,2,3,4,5,0,1,2,3,4,1,2,3,4,5,2,3,4,5,6,3,4,5,6,7},
  {2,1,2,3,4,1,0,1,2,3,2,1,2,3,4,3,2,3,4,5,4,3,4,5,6},
  {3,2,1,2,3,2,1,0,1,2,3,2,1,2,3,4,3,2,3,4,5,4,3,4,5},
  {4,3,2,1,2,3,2,1,0,1,4,3,2,1,2,5,4,3,2,3,6,5,4,3,4},
  {5,4,3,2,1,4,3,2,1,0,5,4,3,2,1,6,5,4,3,2,7,6,5,4,3},
  {2,3,4,5,6,1,2,3,4,5,0,1,2,3,4,1,2,3,4,5,2,3,4,5,6},
  {3,2,3,4,5,2,1,2,3,4,1,0,1,2,3,2,1,2,3,4,3,2,3,4,5},
  {4,3,2,3,4,3,2,1,2,3,2,1,0,1,2,3,2,1,2,3,4,3,2,3,4},
  {5,4,3,2,3,4,3,2,1,2,3,2,1,0,1,4,3,2,1,2,5,4,3,2,3},
  {6,5,4,3,2,5,4,3,2,1,4,3,2,1,0,5,4,3,2,1,6,5,4,3,2},
  {3,4,5,6,7,2,3,4,5,6,1,2,3,4,5,0,1,2,3,4,1,2,3,4,5},
  {4,3,4,5,6,3,2,3,4,5,2,1,2,3,4,1,0,1,2,3,2,1,2,3,4},
  {5,4,3,4,5,4,3,2,3,4,3,2,1,2,3,2,1,0,1,2,3,2,1,2,3},
  {6,5,4,3,4,5,4,3,2,3,4,3,2,1,2,3,2,1,0,1,4,3,2,1,2},
  {7,6,5,4,3,6,5,4,3,2,5,4,3,2,1,4,3,2,1,0,5,4,3,2,1},
  {4,5,6,7,8,3,4,5,6,7,2,3,4,5,6,1,2,3,4,5,0,1,2,3,4},
  {5,4,5,6,7,4,3,4,5,6,3,2,3,4,5,2,1,2,3,4,1,0,1,2,3},
  {6,5,4,5,6,5,4,3,4,5,4,3,2,3,4,3,2,1,2,3,2,1,0,1,2},
  {7,6,5,4,5,6,5,4,3,4,5,4,3,2,3,4,3,2,1,2,3,2,1,0,1},
  {8,7,6,5,4,7,6,5,4,3,6,5,4,3,2,5,4,3,2,1,4,3,2,1,0}};

mat2 = {{0,3,2,0,0,10,5,0,5,2,0,0,2,0,5,3,0,1,10,0,2,1,1,1,0},
  {3,0,4,0,10,0,0,2,2,1,5,0,0,0,0,0,1,6,1,0,2,2,5,1,10},
  {2,4,0,3,4,5,5,5,1,4,0,4,0,4,0,3,2,5,5,2,0,0,3,1,0},
  {0,0,3,0,0,0,2,2,0,6,2,5,2,5,1,1,1,2,2,4,2,0,2,2,5},
  {0,10,4,0,0,2,0,0,0,0,0,0,0,0,2,0,0,2,0,5,0,2,1,0,2},
  {10,0,5,0,2,0,10,10,5,10,6,0,0,10,2,10,1,5,5,2,5,0,2,0,1},
  {5,0,5,2,0,10,0,1,3,5,0,0,2,4,5,10,6,0,5,5,5,0,5,5,0},
  {0,2,5,2,0,10,1,0,10,2,5,2,0,3,0,0,0,4,0,5,0,5,2,2,5},
  {5,2,1,0,0,5,3,10,0,5,6,0,1,5,5,5,2,3,5,0,2,10,10,1,5},
  {2,1,4,6,0,10,5,2,5,0,0,1,2,1,0,0,0,0,6,6,4,5,3,2,2},
  {0,5,0,2,0,6,0,5,6,0,0,2,0,4,2,1,0,6,2,1,5,0,0,1,5},
  {0,0,4,5,0,0,0,2,0,1,2,0,2,1,0,3,10,0,0,4,0,0,4,2,5},
  {2,0,0,2,0,0,2,0,1,2,0,2,0,4,5,0,1,0,5,0,0,0,5,1,1},
  {0,0,4,5,0,10,4,3,5,1,4,1,4,0,0,0,2,2,0,2,5,0,5,2,5},
  {5,0,0,1,2,2,5,0,5,0,2,0,5,0,0,2,0,0,0,6,3,5,0,0,5},
  {3,0,3,1,0,10,10,0,5,0,1,3,0,0,2,0,0,5,5,1,5,2,1,2,10},
  {0,1,2,1,0,1,6,0,2,0,0,10,1,2,0,0,0,5,2,1,1,5,6,5,5},
  {1,6,5,2,2,5,0,4,3,0,6,0,0,2,0,5,5,0,4,0,0,0,0,5,0},
  {10,1,5,2,0,5,5,0,5,6,2,0,5,0,0,5,2,4,0,5,4,4,5,0,2},
```

```
{0,0,2,4,5,2,5,5,0,6,1,4,0,2,6,1,1,0,5,0,4,4,1,0,2},
{2,2,0,2,0,5,5,0,2,4,5,0,0,5,3,5,1,0,4,4,0,1,0,10,1},
{1,2,0,0,2,0,0,5,10,5,0,0,0,0,5,2,5,0,4,4,1,0,0,0,0},
{1,5,3,2,1,2,5,2,10,3,0,4,5,5,0,1,6,0,5,1,0,0,0,0,0},
{1,1,1,2,0,0,5,2,1,2,1,2,1,2,0,2,5,5,0,0,10,0,0,0,2},
{0,10,0,5,2,1,0,5,5,2,5,5,1,5,5,10,5,0,2,2,1,0,0,2,0}};
```

First we define a function to permute rows and columns of a matrix. It simply rearranges the matrix so that both rows and columns are reordered according to a given permutation.

Outline of ***permuteMatrix***

1. Input: a square matrix M and a permutation P of the set $\{1,\ldots,n\}$, where n is the dimension of M.
2. Form $\tilde{M}$, the matrix obtained by rearranging rows and columns of M as specified by P.
3. Return $\tilde{M}$.

```
permuteMatrix[mat_, perm_]:=mat[[perm, perm]]
```

We use a small matrix to see how this works.

```
MatrixForm[mat = Array[x, {4,4}]]
```

$$\begin{pmatrix} x[1,1] & x[1,2] & x[1,3] & x[1,4] \\ x[2,1] & x[2,2] & x[2,3] & x[2,4] \\ x[3,1] & x[3,2] & x[3,3] & x[3,4] \\ x[4,1] & x[4,2] & x[4,3] & x[4,4] \end{pmatrix}$$

Now we move rows/columns (4,1,3,2) to positions (1,2,3,4), and observe the result.

```
MatrixForm[permuteMatrix[mat, {4,1,2,3}]]
```

$$\begin{pmatrix} x[4,4] & x[4,1] & x[4,2] & x[4,3] \\ x[1,4] & x[1,1] & x[1,2] & x[1,3] \\ x[2,4] & x[2,1] & x[2,2] & x[2,3] \\ x[3,4] & x[3,1] & x[3,2] & x[3,3] \end{pmatrix}$$

Let us return to the NUG25 problem. Below is an optimal permutation (it is not unique). We remark that the computation that verified the optimality took substantial time and parallel resources.

```
p = {5,11,20,15,22,2,25,8,9,1,18,16,3,6,19,24,21,14,7,10,17,12,4,23,13};
```

We compute the objective function value we obtain from this permutation. As a sort of baseline, we show the result one obtains from applying no permutation. We then compute results of applying several random permutations. This gives some idea of how to gauge the results below.

```
best = Apply[Plus, Flatten[mat1 * permuteMatrix[mat2, p]]]
```

3744

```
baseline = Apply[Plus, Flatten[mat1 * mat2]]
```

4838

```
randomvals = Table[
  perm = Ordering[RandomReal[{0,1},{25}]];
  Apply[Plus, Flatten[mat1 * permuteMatrix[mat2, perm]]], {10}]
```

{4858, 5012, 5380, 5088, 4782, 4994, 5032, 5044, 5088, 5094}

A substantially longer run over random permutations gives an indication of how hard it is to get good results via a naive random search.

```
SeedRandom[1111];
Timing[randomvals = Table[
  perm = Ordering[RandomReal[{0,1},{25}]];
  Total[Flatten[mat1 * permuteMatrix[mat2, perm]]],
  {1000000}];]
```

{449.06, Null}

```
Min[randomvals]
```

4284

4.6.1 Relative Position Indexing for Permutations

We must decide how to make a set of values into a permutation. Our first approach is nearly identical to the ensemble order method we used on the set partition problem. Specifically, we will let the `Ordering` function of a set of real values determine a permutation.

*Outline of **QAP***

1. Input: square matrices M_1 and M_2 each of dimension n, along with parameter settings to pass to `NMinimize`.
2. Form a vector of variables of length n. Give them initial ranges from 0 to 1.

3. Form an objective function that sums the n^2 products of elements of the first matrix and elements of the row-and-column permuted second matrix.

   The permutation is determined by the ordering of values of the variables vector. (Remark: some readers might recognize this as a *matrix inner product* computed via the *matrix trace* of the usual matrix product).

   For improved speed (at the cost of memory) we *memoize* values of the objective function. What that means is we record them once computed, so that recomputation is done by fast lookup. Readers familiar with data structure methods may recognize this as an application of *hashing*.
4. Call `NMinimize` on the objective function, using the above ranges, constraints, and input option settings.
5. Return the minimal value found, along with the permutation that gives rise to that value.

```
QAP[mat1_, mat2_, cp_, it_, sp_, sc_]:=Module[
 {len = Length[mat1], obfunc, obfunc2, vars, x, nmin, vals, rnges},
 vars = Array[x, len];
 rnges = Map[{#, 0, 1}&, vars];
 obfunc[vec : {__Real}]:=obfunc2[Ordering[vec]];

 obfunc2[perm_]:=obfunc2[perm] =
  Total[Flatten[mat1 * permuteMatrix[mat2, perm]]];
 {nmin, vals} = NMinimize[obfunc[vars], rnges, MaxIterations → it,
  Method → {DifferentialEvolution, SearchPoints → sp, CrossProbability → cp,
    ScalingFactor → sc, PostProcess → False}];
 Clear[obfunc2];
 {nmin, Ordering[vars/.vals]}]
```

Again we face the issue that this problem requires nonstandard values for options to the `DifferentialEvolution` method, in order to achieve a reasonable result. While this is regretable it is clearly better than having no recourse at all. The idea behind having `CrossProbability` relatively small is that we do not want many crossovers in mating a pair of vectors. This in turn is because of the way we define a permutation. In particular it is not just values but relative values across the entire vector that give us the permutation. Thus disrupting more than a few, even when mating a pair of good vectors, is likely to give a bad vector. This was also the case with the set partitioning example we encountred earlier.

We saw that the baseline permutation (do nothing) and random permutations tend to be far from optimal, and even a large random sampling will get us only about half way from baseline to optimal. A relatively brief run with "good" values for the algorithm parameters, on the other hand, yields something notably better. (In the next subsection we explicitly show how one might use short *tuning runs* to find such parameter settings.)

```
SeedRandom[11111];
Timing[{min, perm} = QAP[mat1, mat2, .06, 200, 40, .6]]
```

```
{13.5048, {3864.,
  {22, 20, 17, 12, 5, 13, 15, 23, 25, 2, 19, 10, 9,
   8, 4, 1, 7, 6, 16, 18, 24, 21, 14, 3, 11}}}
```

We now try a longer run.

```
SeedRandom[11111];
Timing[{min, perm} = QAP[mat1, mat2, .06, 4000, 100, .6]]
```

```
{394.881, {3884.,
  {15, 20, 19, 10, 13, 22, 1, 16, 7, 4, 9, 25, 6, 23,
   12, 8, 11, 21, 14, 17, 5, 2, 18, 3, 24}}}
```

We learn a lesson here. Sometimes a short run is lucky, and a longer one does not fare as well. We will retry with a different crossover, more iterations, and a larger set of chromosomes.

```
Timing[{min, perm} = QAP[mat1, mat2, .11, 10000, 200, .6]]
```

```
{2186.43, {3826.,
  {5, 2, 18, 11, 4, 12, 25, 8, 14, 24, 17, 3, 16, 6, 21,
   20, 23, 9, 7, 10, 22, 15, 19, 1, 13}}}
```

This result is not bad.

4.6.2 Representing and Using Permutations as Shuffles

The method we now show will generate a permutation as a *shuffle* of a set of integers. We first describe a standard way to shuffle, with uniform probability, a set of n elements. First we randomly pick a number j_1 in the range $\{1,\ldots,n\}$ and, if $j_1 \neq 1$, we swap the first and j_1th elements. We then select at random an element j_2 in the range $\{2,\ldots,n\}$. If $j_2 \neq 2$ we swap the second and j_2th elements. The interested reader can convince him or herself that this indeed gives a uniform random shuffle (in contrast, selecting all elements in the range $\{1,\ldots,n\}$ fails to be uniform).

Our goal, actually, is not directly to generate shuffles, but rather to use them. Each chromosome will represent a shuffle, encoded as above by a set of swaps to perform. So the effective constraint on the first variable is that it be an integer in the range $\{1,\ldots,n\}$, while the second must be an integer in the range $\{2,\ldots,n\}$, and so on (small point: we do not actually require an nth variable, since its value must always be n). We require a utility routine to convert quickly from a shuffle encoding to a simple permutation vector. The code below will do this. We use the `Compile` function of *Mathematica* to get a speed boost.

Outline of ***getPerm***

1. Input: a shuffle S encoded as $n-1$ integers in the range$\{1,\ldots,n\}$, with the jth actually restricted to lie in the subrange $\{j,\ldots,n\}$.
2. Initialize a vector P of length n to be the identity permutation (that is, the ordered list $\{1,\ldots,n\}$).
3. Iterate over S.
4. Swap the jth element of P with the element whose index is the (current) jth element of S.
5. Return P.

```
getPerm = Compile[{{shuffle, _Integer, 1}}, Module[
  {perm, len = Length[shuffle] + 1},
  perm = Range[len];
  Do[perm[[{j, shuffle[[j]]}]] = perm[[{shuffle[[j]], j}]], {j, len - 1}];
  perm]];
```

Okay, maybe that was a bit cryptic. Here is a brief example that will shed light on this process. Say our shuffle encoding for a set of five elements is $\{2,4,5,4\}$. What would this do to permute the set $\{1,2,3,4,5\}$? First we swap elements 1 and 2, so we have $\{2,1,3,4,5\}$. We next swap elements 2 and 4, giving $\{2,4,3,1,5\}$. Then we swap elements 3 and 5 to obtain $\{2,4,5,1,3\}$. Finally, as the fourth element in our shuffle is a 4, we do no swap. Let us check that we did indeed get the permutation we claim.

```
getPerm[{2,4,5,4}]
```

$\{2,4,5,1,3\}$

The constraints we would like to enforce are that all chromosome elements be integers, and that the jth such element be between j and the total length inclusive. The bit of code below will show how we might set up such constraints.

```
len = 5;
vars = Array[x, len - 1];
constraints = Prepend[Map[(#[[1]] <= # <= len) &, vars], Element[vars, Integers]]
```

$$\{(x[1]|x[2]|x[3]|x[4]) \in \text{Integers}, 1 \le x[1] \le 5, 2 \le x[2] \le 5, 3 \le x[3] \le 5, 4 \le x[4] \le 5\}$$

There is a small wrinkle. It is often faster not to insist on integrality, but rather to use real numbers and simply round off (or truncate). To get uniform probabilities initially, using rounding, we constrain so that a given variable is at least its minimal allowed integer value minus 1/2, and at most its maximal integer value plus 1/2.

Without further fuss, we give an outline and code for this optimization approach.

Outline of ***QAP2***

1. Input: square matrices M_1 and M_2 each of dimension n, along with parameter settings to pass to NMinimize.
2. Form a vector of variables of length $n-1$. For j in $\{1,\ldots,n-1\}$ constrain the jth variable to lie in the range $\{j-.499\ldots,n+1.499\}$.
3. Form an objective function that sums the n^2 products of elements of the first matrix and elements of the row-and-column permuted second matrix. The variables vector, with entries rounded to nearest integers, may be viewed as a shuffle on a set of n elements. The permutation is determined by invoking `getPerm` on the variables vector.
4. Call `NMinimize` on the objective function, using the above variables, constraints, and input option settings.
5. Return the minimal value found, along with the permutation that gives rise to that value.

```
QAP2[mat1_, mat2_, cp_, it_, sp_]:=Module[
 {len = Length[mat1] - 1, obfunc, vars, x, nmin, vals, constraints},
 vars = Array[x, len];
 constraints = Map[(#[[1]] - .499 <= # <= len + 1.499)&, vars];
 obfunc[vec : {__Real}]:=
  Total[Flatten[mat1 * permuteMatrix[mat2, getPerm[Round[vec]]]]];
 {nmin, vals} = NMinimize[{obfunc[vars], constraints}, vars,
  Method -> {DifferentialEvolution, SearchPoints -> sp, CrossProbability -> cp,
  PostProcess -> False}, MaxIterations -> it, Compiled -> False];
 {nmin, getPerm[Round[vars/.vals]]}]
```

We show a sample tuning run. We keep the number of iterations and number of chromosomes modest, and try cross probabilities between 0.05 and 0.95, at increments of .05.

```
Quiet[Table[{j, First[QAP2[mat1, mat2, j/100, 50, 20]]}, {j, 5, 95, 5}]]
```

```
{{5,4364.},{10,4436.},{15,4538.},{20,4428.},{25,4522.},
 {30,4506.},{35,4518.},{40,4550.},{45,4512.},{50,4456.},
 {55,4530.},{60,4474.},{65,4520.},{70,4412.},{75,4474.},
 {80,4454.},{85,4410.},{90,4314.},{95,4324.}}
```

From this we home in on the region of the larger values since they seem to be consistently a bit better than other values (it is interesting that this is the opposite of what I had found for the relative index positioning approach in the previous subsection). We

now do larger runs to get a better idea of what are the relative merits of these various cross probability parameter settings.

```
Quiet[Table[{j,First[QAP2[mat1,mat2,j/100,80,20]]},{j,87,98,1}]]
```

```
{{87,4298.},{88,4418.},{89,4346.},{90,4314.},{91,4396.},{92,4416.},
 {93,4300.},{94,4308.},{95,4274.},{96,4322.},{97,4282.},{98,4298.}}
```

We will finally try a longer run with cross probability set to 0.975.

```
Quiet[Timing[{min,perm} = QAP2[mat1,mat2,.975,10000,100]]]
```

```
{2590.27,{3814.,
  {5,2,11,22,15,18,25,16,9,1,17,3,6,8,
   19,12,14,7,23,20,24,4,21,10,13}}}
```

This gets us reasonably close to the global minimum with a scant 15 lines of code. While it is mildly more complicated than the 10 line relative position indexing method, it has the advantage that it is slightly less dependent on fine tuning of the cross probability parameter.

4.6.3 Another Shuffle Method

There are other plausible ways to set up permutations, such that they behave in a reasonable manner with6 respect to mutation and mating operations. Here is one such.

We have for our vector a set of integers from 1 to n, the length of the set in question (again we will actually work with reals, and round off to get integers). The range restriction is the only stipulation and in particular it may contain repeats. We associate to it a unique permutation as follows. We initialize a list to contain n zeros. The first element in our list is then set to the first element in the vector. We also have a marker set telling us that that first element is now used. We iterate over subsequent elements in our list, setting them to the corresponding values in vector provided those values are not yet used. Once done with this iteration we go through the elements that have no values, assigning them in sequence the values that have not yet been assigned. This method, which is used in [7], is similar to that of `GeneRepair` [8]. It is also related to a method of [12], although they explicitly alter the recombination (that is, the genotype) rather than the resulting phenotype.

Outline of ***getPerm2***

1. Input: a shuffle S encoded as n integers in the range$\{1,\ldots,n\}$.
2. Create vectors P_1 and P_2 of length n. The first will be for the permutation we create, and the second will mark as "used" those elements we have encountered. Initialize elements of each to be 0.

3. Loop over S. Denote by k the jth element of S. If the kth element of P_2 is 0, this means we have not yet used k in our permutation.
   Set $P_2(k)$ to j to mark it as used.
   Set $P_1(j)$ to k.
4. Initialize a counter k to 1.
5. Loop over P_1. If the jth element, $P_1(j)$, is 0 then it needs to be filled in with a positive integer not yet used.
   Find smallest k for which $P_2(k)$ is 0 (telling us that k is not used as yet in the permutation).
   For that k, set $P_1(j)$ to be k, and mark $P_2(k)$ nonzero (alternatively, could simply increment k so it will not revisit this value).
6. Return P_1.

```
getPerm2 = Compile[{{vec, _Integer, 1}}, Module[
  {p1, p2, len = Length[vec], k}, p1 = p2 = Table[0, {len}];
  Do[k = vec[[j]];
   If[p2[[k]] == 0, p2[[k]] = j; p1[[j]] = k;], {j, len}];
  k = 1;
  Do[If[p1[[j]] == 0, While[p2[[k]] ≠ 0, k++];
   p1[[j]] = k;
   p2[[k]] = j], {j, len}];
  p1]];
```

We illustrate with a small example. Say we have the vector $\{4,1,4,3,1\}$. What permutation does this represent? Well, we have a 4 in the first slot, so the resulting permutation vector starts with 4. Then we have a 1, so that's the next element in the permutation. Next is a 4, which we have already used. We defer on that slot. Next is a 3, so the fourth slot in our permutation is 3. last is a 1, which we have already encountered, so we defer on filling in the fifth position of our permutation. We have completed one pass through the permutation. The entries we were unable to use were in positions 3 and 5. The values not yet used are 2 and 5 (because we filled in a vector as $\{4,1,x,3,y\}$, where x and y are not yet known). We now simply use these in order, in the empty slots. That is, entry 3 is 2 and entry 5 is 5. We obtain as our permutation $\{4,1,2,3,5\}$.

```
getPerm2[{4, 1, 4, 3, 1}]
```

$\{4,1,2,3,5\}$

This notion of associating a list with repeats to a distinct shuffle has a clear drawback insofar as earlier elements are more likely than later ones to be assigned to their corresponding values in the vector. All the same, this provides a reasonable way to make a chromosome vector containing repeats correspond to a permutation (and once the method has started to produce permutations, mating/mutation will not cause too many repeats provided the crossover probability is either fairly low or fairly high). Moreover, one can see that any sensible mating process of two chromosomes will less drastically

alter the objective function than would be the case in the ensemble ordering, as the corresponding permutation now depends far less on overall ordering in the chromosomes. The advantage is that this method will thus be somewhat less in need of intricate tuning for the crossover probability parameter (but we will do that anyway).

Outline of ***QAP3***

1. Input: square matrices M_1 and M_2 each of dimension n, along with parameter settings to pass to `NMinimize`.
2. Form a vector of variables of length n. Constrain each variable to lie in the range $\{.501\ldots, n+.499\}$.
3. Form an objective function that sums the n^2 products of elements of the first matrix and elements of the row-and-column permuted second matrix.

   The variables vector, with entries rounded to nearest integers, may be viewed as a shuffle on a set of n elements. The permutation is determined by invoking `getPerm2` on the variables vector.
4. Call `NMinimize` on the objective function, using the above variables, constraints, and input option settings.
5. Return the minimal value found, along with the permutation that gives rise to that value.

```
QAP3[mat1_, mat2_, cp_, it_, sp_]:=Module[
 {len = Length[mat1], obfunc, vars, x, nmin, vals, constraints},
 vars = Array[x, len];
 constraints = Map[(.501 ≤ # ≤ len + 0.499)&, vars];
 obfunc[vec : {__Real}]:=
  Total[Flatten[mat1 * permuteMatrix[mat2, getPerm2[Round[vec]]]]];
 {nmin, vals} = NMinimize[{obfunc[vars], constraints}, vars,
  Method → {DifferentialEvolution, SearchPoints → sp,
    CrossProbability → cp, PostProcess → False},
  MaxIterations → it, Compiled → False];
 {nmin, getPerm2[Round[vars/.vals]]}]
```

We'll start with a tuning run.

```
Quiet[Table[{j, First[QAP3[mat1, mat2, j/100, 50, 20]]}, {j, 5, 95, 5}]]
```

```
{{5,4486.},{10,4498.},{15,4464.},{20,4492.},{25,4430.},
 {30,4516.},{35,4482.},{40,4396.},{45,4432.},{50,4472.},
 {55,4548.},{60,4370.},{65,4460.},{70,4562.},{75,4398.},
 {80,4466.},{85,4378.},{90,4426.},{95,4354.}}
```

I did a second run (not shown), in the upper range of crossover probabilities, and with more iterations and larger numbers of search points. It homed in on .93 as a reasonably good choice for a crossover probability setting.

```
Timing[Quiet[QAP3[mat1, mat2, .93, 8000, 100]]]

{2380.2, {3888.,
  {7, 20, 11, 8, 13, 4, 25, 10, 19, 18, 17, 22, 6, 3, 5, 15, 24,
   14, 23, 21, 1, 16, 2, 12, 9, 26}}}
```

4.7 Hybridizing Differential Evolution for the Assignment Problem

Thus far we have seen methods that, for a standard benchmark problem from the quadratic assignment literature, take us to within shouting distance of the optimal value. These methods used simple tactics to formulate permutations from a vector chromosome, and hence could be applied within the framework of Differential Evolution. We now show a method that hybridizes Differential Evolution with another approach.

A common approach to combinatorial permutation problems is to swap pairs (this is often called *2-opt*), or reorder triples, of elements (also reversal of segments is common). With Differential Evolution one might do these by modifying the objective function to try them, and then recording the new vector (if we choose to use it) in the internals of the algorithm. This can be done in `NMinimize`, albeit via alteration of an entirely undocumented internal variable. We show this below, using a simple set of pair swaps. When we obtain improvement in this fashion, we have gained something akin to a local *hill climbing* method. I remark that such hybridization, of an evolutionary method with a local improvement scheme, is often referred to as a *memetic* algorithm. Nice expositions of such approaches can be found in [9] and [6].

The code creates a random value to decide when to use a swap even if it resulted in no improvement. This can be a useful way to maintain variation in the chromosome set. We also use a print flag: if set to `True`, whenever we get an improvement on the current best permutation, we learn what is the new value and how much time elapsed since the last such improvement. We also learn when we get such an improvement arising from a local change (that is, a swap).

As an aside, the use of a swap even when it gives a worse result has long standing justification. The idea is that we allow a decrease in quality in the hope that it will later help in finding an improvement. This is quite similar to the method of *simulated annealing*, except we do not decrease the probability, over the course of generations, of accepting a decrease in quality.

Outline of ***QAP4***

1. Input: square matrices M_1 and M_2 each of dimension$n$, along with parameter settings to pass to `NMinimize`, and a probability level p between 0 and 1 to determine when to retain an altered chromosome that gives a decrease in quality.
2. Form a vector of variables of length n.

3. Give them initial ranges from 0 to 1.
4. Form an objective function that sums the n^2 products of elements of the first matrix and elements of the row-and-column permuted second matrix. As in `QAP`, the permutation is determined by the ordering of values of the variables vector.
5. Iterate some number of times (a reasonable value is 4).
   Swap a random pair of elements in the variables vector.
   Check whether we got improvement in the objective function.
   If so, keep this improved vector.
   If not, possibly still keep it depending on whether a random value between 0 and 1 is larger than p, and also whether the better vector is the best seen thus far (we never replace the best one we have).
   Depending on an input flag setting, either restart the swapping (if we are not done iterating) with our original vector, or else continue with the one created from prior swaps.
6. Call `NMinimize` on the objective function, using the above ranges, constraints, and input option settings.
7. Return the minimal value found, along with the permutation that gives rise to that value.

```
QAP4[mat1_, mat2_, cp_, it_, sp_, sc_, maxj_:4, keep_:0.4, restorevector_,
 printFlag_:False]:=Module[
 {len = Length[mat1], objfunc, objfunc2, vars, vv, nmin, vals, rnges, best,
  bestvec, indx = 0, i = 0, tt = TimeUsed[]},
 vars = Array[vv, len];
 rnges = Map[{#, 0, 1}&, vars];
 objfunc2[vec_]:=objfunc2[vec] =
 Total[Flatten[mat1 * permuteMatrix[mat2, vec]]];
 objfunc[vec : {__Real}]:=Module[
  {val1, val2, r1, r2, vec1 = vec, vec2 = vec, max = Max[Abs[vec]], j = 0},
  {vec1, vec2} = {vec1, vec2}/max;
  val1 = objfunc2[Ordering[vec1]];
  While[j ≤ maxj,
  j++;
  {r1, r2} = RandomInteger[{1, len}, {2}];
  If[restorevector, vec2 = vec1];
  vec2[[{r1, r2}]] = vec2[[{r2, r1}]];
  val2 = objfunc2[Ordering[vec2]];
  If[val2 < best, j–;
   If[printFlag, Print["locally improved", {best, val2}]]];
   If[val2 ≤ val1||(val1 > best&&RandomReal[] > keep),
    OptimizeNMinimizeDump`vec = vec2;
    If[val2 < val1, vec1 = vec2];
```

```
    val1 = Min[val1, val2],
    OptimizeNMinimizeDump`vec = vec1];
   If[val1 < best,
    best = val1;
    vec1 = bestvec = OptimizeNMinimizeDump`vec;
    If[printFlag,
     Print["new low ", ++indx, " {iteration, elapsedtime, newvalue} ",
     {i, TimeUsed[] - tt, best}]]; tt = TimeUsed[];];
   ];
  val1];
 bestvec = Range[len];
 best = Total[Flatten[mat1 * mat2]];
 {nmin, vals} = NMinimize[objfunc[vars], rnges,
   MaxIterations → it, Compiled → False, StepMonitor :→ i++,
  Method → {DifferentialEvolution, SearchPoints → sp, ,
    CrossProbability → cpScalingFactor → sc, PostProcess → False}];
 Clear[objfunc2];
 {Total[Flatten[mat1 * permuteMatrix[mat2,
    Ordering[bestvec]]]], Ordering[bestvec]}]
```

We now show a run with printout included. The parameter settings are, as usual, based on shorter tuning runs.

```
Timing[QAP4[mat1, mat2, .08, 400, 320, .4, 4, .4, False, True]]
```

```
locally improved{4838, 4788}
new low 1 {iteration, elapsed time, new value} {0, 0.280017, 4788}
locally improved{4788, 4724}
new low 2 {iteration, elapsed time, new value} {0, 0.012001, 4724}
locally improved{4724, 4696}
new low 3 {iteration, elapsed time, new value} {0, 0., 4696}
locally improved{4696, 4644}
new low 4 {iteration, elapsed time, new value} {0, 0., 4644}
locally improved{4644, 4612}
new low 5 {iteration, elapsed time, new value} {0, 0.240015, 4612}
locally improved{4612, 4594}
new low 6 {iteration, elapsed time, new value} {0, 0.100006, 4594}
locally improved{4594, 4566}
new low 7 {iteration, elapsed time, new value} {0, 0.004, 4566}
locally improved{4566, 4498}
new low 8 {iteration, elapsed time, new value} {0, 0., 4498}
locally improved{4498, 4370}
new low 9 {iteration, elapsed time, new value} {0, 0.972061, 4370}
locally improved{4370, 4348}
new low 10 {iteration, elapsed time, new value} {0, 0.004, 4348}
locally improved{4348, 4322}
new low 11 {iteration, elapsed time, new value} {10, 21.3933, 4322}
```

```
new low 12 {iteration, elapsed time, new value} {11, 0.96806, 4308}
new low 13 {iteration, elapsed time, new value} {20, 19.0252, 4304}
locally improved{4304, 4242}
new low 14 {iteration, elapsed time, new value} {20, 1.88812, 4242}
locally improved{4242, 4184}
new low 15 {iteration, elapsed time, new value} {22, 4.29227, 4184}
new low 16 {iteration, elapsed time, new value} {29, 14.7769, 4174}
new low 17 {iteration, elapsed time, new value} {31, 4.57229, 4102}
locally improved{4102, 4096}
new low 18 {iteration, elapsed time, new value} {37, 12.3448, 4096}
locally improved{4096, 4092}
new low 19 {iteration, elapsed time, new value} {41, 8.0405, 4092}
new low 20 {iteration, elapsed time, new value} {51, 22.2414, 4082}
new low 21 {iteration, elapsed time, new value} {55, 8.28452, 4076}
new low 22 {iteration, elapsed time, new value} {56, 3.51622, 4072}
new low 23 {iteration, elapsed time, new value} {56, 0.396025, 3980}
new low 24 {iteration, elapsed time, new value} {62, 13.1488, 3964}
new low 25 {iteration, elapsed time, new value} {64, 3.03619, 3952}
locally improved{3952, 3948}
new low 26 {iteration, elapsed time, new value} {71, 16.385, 3948}
new low 27 {iteration, elapsed time, new value} {75, 8.38452, 3940}
new low 28 {iteration, elapsed time, new value} {78, 6.30839, 3934}
new low 29 {iteration, elapsed time, new value} {85, 14.0169, 3930}
new low 30 {iteration, elapsed time, new value} {85, 0.980061, 3924}
new low 31 {iteration, elapsed time, new value} {86, 1.71611, 3922}
locally improved{3922, 3894}
new low 32 {iteration, elapsed time, new value} {89, 7.22845, 3894}
locally improved{3894, 3870}
new low 33 {iteration, elapsed time, new value} {109, 42.1226, 3870}
new low 34 {iteration, elapsed time, new value} {119, 22.5814, 3860}
new low 35 {iteration, elapsed time, new value} {134, 33.4381, 3856}
locally improved{3856, 3840}
new low 36 {iteration, elapsed time, new value} {142, 16.269, 3840}
new low 37 {iteration, elapsed time, new value} {146, 8.72855, 3830}
new low 38 {iteration, elapsed time, new value} {174, 57.7716, 3816}
new low 39 {iteration, elapsed time, new value} {196, 44.6508, 3800}
new low 40 {iteration, elapsed time, new value} {203, 13.5768, 3788}
new low 41 {iteration, elapsed time, new value} {203, 0.400025, 3768}
locally improved{3768, 3750}
new low 42 {iteration, elapsed time, new value} {222, 34.3741, 3750}

{590.045, {3750,
  {1, 19, 22, 15, 13, 7, 10, 9, 20, 23, 21, 6, 14, 4, 17, 16, 3, 8, 25, 12, 24, 18, 11, 2, 5}}}
```

This is now quite close to the global minimum. As might be observed from the printout, the swaps occasionally let us escape from seemingly sticky local minima. So,

for the problem at hand, this hybridization truly appears to confer an advantage over pure Differential Evolution. I will remark that it seems a bit more difficult to get this type of hybridization to cooperate well with the various shuffle methods of creating permutations.

For contrast we go to the opposite extreme and do a huge number of swaps, on a relatively smaller number of chromosomes and using far fewer iterations. We will reset our vector with swapped pairs to the original (or best variant found thereof, if we get improvements). This is to avoid straying far from reasonable vectors, since we now do many swaps.

This is thus far a 2-opt approach rather than Differential Evolution per se. Nonetheless, we notice that the later stages of improvement do come during the actual iterations of Differential Evolution, and quite possibly those final improvements are due in part to the maintaining of diversity and the use of mutation and recombination.

Timing[QAP4[mat1, mat2, .08, 20, 60, .4, 2000, .6, True, True]]

locally improved{4838, 4808}
new low 1 {iteration, elapsed time, new value} {0, 0.048003, 4808}
locally improved{4808, 4786}
new low 2 {iteration, elapsed time, new value} {0, 0.004001, 4786}
locally improved{4786, 4738}
new low 3 {iteration, elapsed time, new value} {0, 0., 4738}
locally improved{4738, 4690}
new low 4 {iteration, elapsed time, new value} {0, 0.004, 4690}
locally improved{4690, 4614}
new low 5 {iteration, elapsed time, new value} {0, 0., 4614}
locally improved{4614, 4502}
new low 6 {iteration, elapsed time, new value} {0, 0., 4502}
locally improved{4502, 4406}
new low 7 {iteration, elapsed time, new value} {0, 0., 4406}
locally improved{4406, 4370}
new low 8 {iteration, elapsed time, new value} {0, 0., 4370}
locally improved{4370, 4342}
new low 9 {iteration, elapsed time, new value} {0, 0.004, 4342}
locally improved{4342, 4226}
new low 10 {iteration, elapsed time, new value} {0, 0., 4226}
locally improved{4226, 4178}
new low 11 {iteration, elapsed time, new value} {0, 0.016001, 4178}
locally improved{4178, 4174}
new low 12 {iteration, elapsed time, new value} {0, 0., 4174}
locally improved{4174, 4170}
new low 13 {iteration, elapsed time, new value} {0, 0.016001, 4170}
locally improved{4170, 4158}
new low 14 {iteration, elapsed time, new value} {0, 0.012001, 4158}
locally improved{4158, 4114}
new low 15 {iteration, elapsed time, new value} {0, 0.004, 4114}

locally improved{4114, 4070}
new low 16 {iteration, elapsed time, new value} {0, 0.004, 4070}
locally improved{4070, 4046}
new low 17 {iteration, elapsed time, new value} {0, 0.016001, 4046}
locally improved{4046, 4042}
new low 18 {iteration, elapsed time, new value} {0, 0.060004, 4042}
locally improved{4042, 4014}
new low 19 {iteration, elapsed time, new value} {0, 0.080005, 4014}
locally improved{4014, 3982}
new low 20 {iteration, elapsed time, new value} {0, 0.008001, 3982}
locally improved{3982, 3978}
new low 21 {iteration, elapsed time, new value} {0, 0.008, 3978}
locally improved{3978, 3970}
new low 22 {iteration, elapsed time, new value} {0, 0.052003, 3970}
locally improved{3970, 3966}
new low 23 {iteration, elapsed time, new value} {0, 0.096006, 3966}
locally improved{3966, 3964}
new low 24 {iteration, elapsed time, new value} {0, 0.012001, 3964}
locally improved{3964, 3960}
new low 25 {iteration, elapsed time, new value} {0, 0., 3960}
locally improved{3960, 3944}
new low 26 {iteration, elapsed time, new value} {0, 0.032002, 3944}
locally improved{3944, 3926}
new low 27 {iteration, elapsed time, new value} {0, 0.036002, 3926}
locally improved{3926, 3916}
new low 28 {iteration, elapsed time, new value} {0, 0.004001, 3916}
locally improved{3916, 3896}
new low 29 {iteration, elapsed time, new value} {0, 0.032002, 3896}
locally improved{3896, 3892}
new low 30 {iteration, elapsed time, new value} {0, 0.112007, 3892}
locally improved{3892, 3888}
new low 31 {iteration, elapsed time, new value} {0, 0.096006, 3888}
locally improved{3888, 3868}
new low 32 {iteration, elapsed time, new value} {0, 0.104006, 3868}
locally improved{3868, 3864}
new low 33 {iteration, elapsed time, new value} {0, 2.18414, 3864}
locally improved{3864, 3860}
new low 34 {iteration, elapsed time, new value} {0, 0.116007, 3860}
locally improved{3860, 3852}
new low 35 {iteration, elapsed time, new value} {0, 1.84411, 3852}
locally improved{3852, 3838}
new low 36 {iteration, elapsed time, new value} {0, 0.028002, 3838}
locally improved{3838, 3834}
new low 37 {iteration, elapsed time, new value} {0, 0.016001, 3834}
locally improved{3834, 3818}

```
new low 38 {iteration, elapsed time, new value} {0,0.072004,3818}
locally improved{3818,3812}
new low 39 {iteration, elapsed time, new value} {0,3.55622,3812}
locally improved{3812,3786}
new low 40 {iteration, elapsed time, new value} {0,0.084006,3786}
locally improved{3786,3780}
new low 41 {iteration, elapsed time, new value} {0,1.6361,3780}
locally improved{3780,3768}
new low 42 {iteration, elapsed time, new value} {0,0.048003,3768}
locally improved{3768,3758}
new low 43 {iteration, elapsed time, new value} {0,0.096006,3758}
locally improved{3758,3756}
new low 44 {iteration, elapsed time, new value} {7,315.692,3756}
locally improved{3756,3754}
new low 45 {iteration, elapsed time, new value} {10,108.415,3754}
locally improved{3754,3752}
new low 46 {iteration, elapsed time, new value} {10,0.15601,3752}
locally improved{3752,3748}
new low 47 {iteration, elapsed time, new value} {16,245.787,3748}
locally improved{3748,3744}
new low 48 {iteration, elapsed time, new value} {16,0.076005,3744}

{872.687,{3744,
  {22,15,20,11,5,1,9,8,25,2,19,6,3,16,18,10,7,14,21,24,13,23,4,12,17}}}
```

Notice that this permutation is not identical to the one we presented at the outset, which in turn comes from benchmark suite results in the literature. Also note that we seem to get good results from swaps early on (indeed, we almost get a global minimizer prior to the main iterations). This raises the question of whether it might be useful to plug in a different sort of heuristic, say larger swaps, or perhaps use of local (continuous) quadratic programming. The interested reader may wish to explore such possibilities.

4.8 Future Directions

We have seen several examples of discrete optimization problems, and indicated ways in which one might approach them using Differential Evolution. Problems investigated include basic integer programming, set partitioning, set covering by subsets, and the common permutation optimization problem of quadratic assignment. The main issues have been to adapt Differential Evolution to enforce discrete or combinatorial structure, e.g. that we obtain integrality, partitions, or permutations from chromosome vectors.

There are many open questions and considerable room for development. Here are a few of them.

- Figure out better ways to attack quadratic assignment problems so that we are less likely to encounter difficulty in tuning parameter values, premature convergence, and so on.

- Make the Differential Evolution program *adaptive*, that is, allow algorithm parameters themselves to be modified during the course of a run. This might make results less sensitive to tuning of parameters such as `CrossProbability`.
- Alternatively, develop a better understanding of how to select algorithm parameters in a problem-specific manner. Our experience has been that settings for cross probability should usually be around .9 (which is quite high as compared to what is typical for continuous optimization). It would be useful to have a more refined understanding of this and other tuning issues.
- Figure out how to sensibly alter parameters over the course of the algorithm, not by evolution but rather by some other measure, say iteration count. For example, one might do well to start of with a fairly even crossover (near 0.5, that is), and have it either go up toward 1, or drop toward 0, as the algorithm progresses. Obviously it is not hard to code Differential Evolution to do this. What might be interesting research is to better understand when and how such progression of algorithm parameters could improve performance.
- Implement a two-level version of Differential Evolution, wherein several short runs are used to generate initial values for a longer run.
- Use Differential Evolution in a hybridized form, say, with intermediate steps of local improvement. This would involving modifying chromosomes "in plac", so that improvements are passed along to subsequent generations. We showed a very basic version of this but surely there must be improvements to be found.

We remark that some ideas related to item 2 above are explored in [5]. Issues of self-adaptive tuning of Differential Evolution are discussed in some detail in [1]. A nice exposition of early efforts along these lines, for genetic algorithms, appears in [3].

References

1. Brest, J., Greiner, S., Bošković, B., Mernik, M., Žumer, V.: Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems. IEEE Trans. Evol. Comput. 10, 646–657 (2006)
2. Gisvold, K., Moe, J.: A method for nonlinear mixed-integer programming and its application to design problems. J. ENg. Ind. 94, 353–364 (1972)
3. Goldberg, D.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
4. Goldberg, D., Lingle, R.: Alleles, loci, and the traveling salesman problem. In: Proceedings of the 1st International Conference on Genetic Algorithms, pp. 154–159. Lawrence Erlbaum Associates, Inc., Mahwah (1985)
5. Jacob, C.: Illustrating Evolutionary Computation with Mathematica. Morgan Kaufmann Publishers Inc., San Francisco (2001)
6. Krasnogor, N., Smith, J.: A tutorial for competent memetic algorithms: Model, taxonomy and design issues. IEEE Trans. Evol. Comput. 9, 474–488 (2005)
7. Lichtblau, D.: Discrete optimization using Mathematica. In: Proceedings of the World Conference on Systemics, Cybernetics, and Informatics. International Institute of Informatics and Systemics, vol. 16, pp. 169–174 (2000)

8. Mitchell, G., O'Donoghue, D., Barnes, D., McCarville, M.: GeneRepair: a repair operator for genetic algorithms. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Kendall, G., Wilson, S.W., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A., Dowsland, K.A., Jonoska, N., Miller, J., Standish, R.K. (eds.) GECCO 2003. LNCS, vol. 2724. Springer, Heidelberg (2003)
9. Moscato, P.: On evolution, search, optimization, genetic algorithms, and martial arts. Concurrent Computation Program 826, California Institute of Technology (1989)
10. Price, K., Storn, R.: Differential evolution. Dr. Dobb's Journal 78, 18–24 (1997)
11. Price, K., Storn, R., Lampinen, J.: Differential Evolution: A Practical Approach to Global Optimization. Natural Computing Series. Springer, New York (2005)
12. Tate, D., Smith, A.: A genetic approach to the quadratic assignment problem. Computers and Operations Research 22(1), 73–83 (1995)
13. Wolfram Research, Inc., Champaign, Illinois, USA, Mathematica 6 (2007) (Cited September 25, 2008), `http://reference.wolfram.com/mathematica/ref/NMinimize.html`