Ashdin Publishing
International Journal of Swarm Intelligence and Evolutionary Computation
Vol. 1 (2012), Article ID Z110301, 10 pages
doi:10.4303/ijsiec/Z110301

Research Article

Differential Evolution in Discrete Optimization

Daniel Lichtblau

Wolfram Research, Inc., 100 Trade Center Dr., Champaign, IL 61820, USA
Address correspondence to Daniel Lichtblau, danl@wolfram.com

Received 30 March 2011; Revised 2 June 2012; Accepted 5 June 2012

Abstract We show ways in which differential evolution, a member of the genetic/evolutionary family of global optimization methods, can be used for the purpose of discrete optimization. We consider several nontrivial problems arising from actual practice, using differential evolution as our primary tool to obtain good results. We indicate why methods more commonly seen in discrete optimization, such as integer linear programming, might be less effective for problems of the type we will consider.

Keywords discrete optimization; evolutionary methods; knapsack problem

1 Introduction

Differential evolution (henceforth abbreviated as DE) is a member of the evolutionary algorithms family of optimization methods. It was first introduced by Price and Storn in the 1990s [22]. The primary motivation was to provide a natural way to handle continuous variables in the setting of an evolutionary algorithm; while similar to many genetic methods, it uses continuous rather than discrete variables. A distinguishing feature is that in addition to mating, mutations are also determined from members of the pool. The mutations arise as weighted differences of population members, hence form a type of "population-derived noise" (in the words of Storn and Price).

DE is now a mainstream method for global optimization [1,17,18,27]. It is moreover a fundamental method used in *Mathematica* [25]. Our experience using it as a general purpose method has been quite good. It is adaptable to handling constraints, has flexible tuning parameters (and is generally robust so as to usually spare users the need for altering from default settings thereof). It is, moreover, adaptable to some extent to the setting of discrete and combinatorial optimization [8,10,17] (and similar ideas appear in [7,12, 14,23] with regard to other optimization methods).

In this paper, we will give a brief account of how DE can be gainfully applied to certain classes of discrete and mixed optimization problems. Our goal is to show ways in which DE can be used for discrete and mixed problems, in situations where more classical methods such as integer linear programming might be inapplicable or otherwise undesirable. The problems we discuss all have practical importance in their respective domains.

Here are the specific applications we present:

(1) Find a chemical equation of state given appropriate data.
(2) Solve a knapsack problem relevant to computer file storage.
(3) Minimize a certain column sum in a sparse matrix, with application to categorizing language queries.
(4) Schedule jobs according to some given criteria, in such a way as to be reasonably "robust" (e.g. to delay).

Our examples come from disparate fields but share an important feature: they neither fit well into the world of classical optimization methods, nor, seemingly, are well matched to evolutionary methods. The main contribution of this paper is to show how, using careful formulation of objectives and constraints, one may obtain good results with DE on problems of these types.

2 Brief review of differential evolution

We first recall the general setup for DE [18,22]. We have some number of vectors, or *chromosomes*, of continuous-valued genes. Each will mate with a randomly chosen different vector according to a crossover probability, mutated by the weighted difference of a distinct pair of different chromosomes in the pool. We remark that one gene, selected at random, *must* be altered by crossover. The child thus formed will compete with its main parent chromosome (the one that was not chosen at random) to determine which is to be retained for the next generation. All these steps are as described by Price and Storn [22].Variations of the algorithm use differing probability distributions for the random parameters. Some also use the best vector in the population, rather than a randomly chosen one, as a basis for all mating [18,22].

In the terminology of the literature, *Mathematica* uses the classical DE/rand/1/bin [18,22,27] strategy (i.e., random choice of other parent, crossover determined by as binary variable, and one pair used for mutations). Random parameters all come from the uniform distribution over their respective allowable ranges; this was what was used in [22]. For the crossover parameter this equates to use of a binomial

distribution. We remark that some variants of DE instead use a crossover that places one contiguous subsequence from the secondary parent into the chromosome of the main parent. The locus point is chosen at random and the length is a random variate that approximates an exponential distribution. See [18,27] for further details.

DE has parameters to determine crossover probability, mutation weighting, number of generations to allow, and number of chromosomes per generation. The *Mathematica* function that is used for DE is called `NMinimize`. We describe some details in the code appendix, for example how one sets the various algorithm parameters described above.

Since DE was originally formulated in the context of continuous optimization, and without much detail regarding constraints, we should describe how equality/inequality constraints are handled and how integrality is enforced. The first is fairly standard. Specifically, inequality constraints are enforced in one of two ways. For upper and lower bounds on individual variables (i.e., box-type constraints), variables that exceed their constraints will be moved back into the range, or allowed to remain on the boundary closer to the violation, depending on how far along the algorithm has progressed. Other inequality constraints are handled with penalty functions. These are at first relatively lax, and increased over generations. This is similar in principle, though not specifics, to the adaptive penalty method of [3]. Linear equalities are handled by elimination of variables (according to a heuristic based on singular values decomposition, so as to avoid near linear dependencies of the remaining variables). Nonlinear equalities are treated as two-sided inequalities, again using penalties.

Next is the question of how integrality is enforced. This is particularly important since we will need to constrain variables to take on only integer values in specific ranges. For this 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 [4]. For example, one could add, for each variable x, a penalty term of the form $(x - \mathrm{round}(x))^2$ (perhaps multiplied by some suitably large constant). `NMinimize` does not use this approach, but a user can enable it to do so by explicitly using the `"PenaltyFunction"` method option. Another method, the one actually 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).

While the main focus of DE has been for continuous optimization, there is some prior work applying it to discrete and even combinatorial optimization problems. See [8,17], and especially [10] and references therein. The main goal of present work is to add to the overall understanding of the viable uses of DE in the realm of discrete optimization.

We now proceed to our specific examples.

3 Chemical equation of state (a mixed discrete-continuous optimization)

We first work out an example that was brought to my attention by Lai Ngoc Anh (private communication). It arises in the context of computational chemistry. We have numerical data and wish to find a good equation of state approximation. This will be a Puisiux polynomial function (i.e., exponents may be rational instead of integer) of modest degree, with nonnegative exponents. Moreover we seek a sparse polynomial (customarily referred to as a fewnomial). We will need to find both the exponents and coefficients.

This is amenable to handling as a bilevel optimization problem. At the outer level we attempt to find good exponents in a discrete optimization. With exponents fixed we then proceed to solve a continuous optimization in order to find good corresponding coefficients. That is, the outer level of optimization will use DE to select integer values for the exponent variables, and the inner level then optimizes the coefficients for a particular choice of exponents. The objective being optimized at that stage is the sum of squared discrepancies. We will use DE for the exponent optimization, as it readily allows us to enforce integrality. The inner (coefficient-finding) stage is handled by local methods as they are both fast and adequate for that task. So the strategy is to call on DE with an objective function that is itself computed by an optimization function.

The data are modified (differently scaled) from [15, Table 1]. A nice article describing how such a fit was earlier performed is [20]. Our method can be viewed as a simpler variant; all the hard work is left to the DE implementation. As an added benefit, we will emerge with a slightly better result than that earlier one.

The actual input data and implementation is found in the code appendix. It is configured to let the user decide how many generations (iterations) to allow the optimizer to run, and also to provide the number of points in the search space (in DE this remains constant across generations). Our inner objective function will be a sum of squares of discrepancies between calculated and actual values. We explicitly allow for a certain number of nonzero coefficient/exponent pairs. We could (but do not) constrain the maximal exponent values as well. Likewise, we could allow for negative exponents, if we thought they might be useful in the result we seek; we do not do so in this instance. As a generalization, our actual code also allows for inputs involving more than one independent variable; the result will then be a multivariate Puisiux polynomial. This can be useful, e.g. for approximate recovery of a fewnomial from a set of multivariate data values. One might also generalize in other directions, e.g. by allowing for imposing constraints on total degree of monomials or positivity on coefficients. We do not pursue these here, but only mention that such constraints are quite simple to add.

We include the possibility of printing some internal information so that one can see what actually is the objective function we seek to minimize. For purposes of brevity we show this on a much smaller data set, with a fewnomial comprised of but two terms. Our actual run will remove these restrictions and proceed with this printing option disabled.

The structure of a function call is

```
findFewnomial[data_,vars_,nterms_,iters_,
              npts_,den_:1,prnt_:False,
              lvars_:{c,j}]
```

The first argument is simply a list of data values, where each element is itself a (nested) list of the form

```
{{independentvariablevalues},functionvalue}.
```

The second argument is the list of independent variable names. Third is the number of terms we allow in our resulting fewnomial. We then give the number of generations and the number of chromosomes in each generation. We next provide a common denominator for the exponents, should we want to allow fractional values (the point here is to be able to work internally with strictly integer values). The default value for this denominator is 1. Last are arguments used for pedagogy, specifically whether to print intermediate things as below, and, if so, what symbols to use to denote coefficients and exponents.

Our very simple test case uses four values from the full table, and seeks to fit a fewnomial of two terms, with exponent denominator 6, to those values. Our objective function will thus be of the form

$$\sum_{j=1}^{4} \left(c_1 {x_j}^{e_1} + c_2 {x_j}^{e_2} - y_j\right)^2,$$

where the pairs (x_j, y_j) come from the actual data, and we must find suitable values for the exponents (e_1, e_2) and coefficients (c_1, c_2).

```
findFewnomial[Take[data2,4],{x},
              2,20,5,6,True]
```

polynomial is

$$c(1)x^{\frac{1}{6}j(1,1)} + c(2)x^{\frac{1}{6}j(1,2)}$$

sum of squared discrepancies is

$$\begin{aligned}&(c(1)0.461139^{\frac{1}{6}j(1,1)} + c(2)0.461139^{\frac{1}{6}j(1,2)} + 2.57627)^2\\ &+ (c(1)0.476987^{\frac{1}{6}j(1,1)} + c(2)0.476987^{\frac{1}{6}j(1,2)} + 2.66935)^2\\ &+ (c(1)0.484912^{\frac{1}{6}j(1,1)} + c(2)0.484912^{\frac{1}{6}j(1,2)} + 2.71628)^2\\ &+ (c(1)0.492836^{\frac{1}{6}j(1,1)} + c(2)0.492836^{\frac{1}{6}j(1,2)} + 2.7635)^2\end{aligned}$$

$$\{1.9712810185785484` *^ \text{-}7, 0.425061\sqrt{x} - 6.21231x\}.$$

The result indicates that our sum of squares residual is on the order of 10^{-7} after a run of 20 generations using but 5 chromosomes, and our optimal fewnomial is comprised of a square root and a linear term.

We now invoke this with a specification to run for up to 200 generations, using 50 search points (and of course the full data set). We look for a Puisiux polynomial of four terms, where denominators are 6. These choices are based on domain-specific knowledge outside the scope of this article.

```
Timing[findFewnomial[data2,{x},4,200,50,6]]
```

$$\begin{aligned}&\{171.079, \{3.7037838494212757` *^ -9, 3.7723x^{5/3}\\ &- 1.87827x^{14/3} - 3.08831x^{11/6} - 6.10819x\}\}\end{aligned}$$

The first part of the result indicates run time in seconds. Next is the optimal value found for the objective function, followed by the Puisiux polynomial that gives that optimum. We have obtained a rather low sum of squares of discrepancies, using a scant two dozen or so lines of code.

We remark that very similar ideas can be used to find sparse polynomial approximate GCDs (i.e., polynomials that are in some measure approximate common divisors of a given polynomial set, and have maximal degree). Similarly one can use these techniques to find approximate factorizations. Such approximations comprise an area of active interest in symbolic-numeric computation.

4 File storage (a knapsack problem)

This next problem was communicated by Peter Sisak, in the Usenet forum comp.soft-sys.math.mathematica [21]. We are given a set of file sizes in bytes. We have a storage device of a given size and wish to store as much, in total bytes, of the given files as possible. Our constraints are that we cannot exceed the storage limit, and we will not split files (so we cannot fill the device using a fraction of a final file).

One will observe that this is a traditional knapsack problem. As such, one might seek to handle it using standard integer linear programming tools such as a branch-and-bound strategy. The actual sizes given, unfortunately, render such an approach quite problematic. Indeed, the method discussed in [9] did not terminate in reasonable time, while code from [11] terminated, but not with a plausible result. There are various ways to ameliorate this, e.g by dividing and finding only an approximate best solution. Some approaches along these lines are shown in responses to the original query; see the URL in [21] for further details. We will instead use DE to find a reasonable approximation, as this requires no preprocessing, and but a few lines of code.

We set up the problem quite simply. We have a set of 0-1 variables, where a value of 1 means we will use the corresponding file and 0 indicates we will not. Our objective is to minimize the difference between the total size we use and

the target (i.e., the size of our storage device). We constrain so that the discrepancy is positive, that is, we do not exceed the limit.

As before, we will first illustrate on a small random problem. We turn on the printing of intermediate parts to show, explicitly, the objective function and constraints. In this small example we use a total size of 2000 and start with five sizes randomly chosen from the range $(0,1000)$: the actual ones used are $(270,66,603,811,857)$. We will allow up to 80 generations, using 20 chromosomes. The crossover probability is set to its default of 0.5.

```
SeedRandom[11111];
closestSum[RandomInteger[{1000},5],
           2000,80,20,.5,True]
```

objective: $270\mathrm{x}(1)+66\mathrm{x}(2)+603\mathrm{x}(3)+811\mathrm{x}(4)+857\mathrm{x}(5)$

$$\text{constraints:}\begin{array}{l} 0\le \mathrm{x}(1)\le 1\\ 0\le \mathrm{x}(2)\le 1\\ 0\le \mathrm{x}(3)\le 1\\ 0\le \mathrm{x}(4)\le 1\\ 0\le \mathrm{x}(5)\le 1\\ 270\mathrm{x}(1)+66\mathrm{x}(2)+603\mathrm{x}(3)+811\mathrm{x}(4)\\ \qquad +857\mathrm{x}(5)\le 2000\\ (\mathrm{x}(1)|\mathrm{x}(2)|\mathrm{x}(3)|\mathrm{x}(4)|\mathrm{x}(5))\in\mathbb{Z}\end{array}$$

$\{1938.,62.,0.031,\{1,0,0,1,1\}\}$.

As stated, this is clearly the setup of a 0-1 knapsack problem. The optimal choice in this small example uses the first, fourth, and fifth values from the given set.

We now run this on the example of interest. We will allow 300 iterations and use a search space of 100 chromosomes.

```
Timing[Quiet[closestSum[bigvals,targ,
                        300,100]]]
```

$\{109.348,\{8.54792\times10^{9},73942,8.650217052104485`*^-6,$
$\{0,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,$
$0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,$
$0,0,1,0,0,0,0,1,1,0,0\}\}\}$

While this can be shown to be less than the optimum, we see we that for all intents and purposes we have attained our practical goal. That is to say, the fraction of remaining storage, around 10^{-5}, is miniscule relative to that which will be utilized from the computed set of files sizes. And the remaining amount, 73942, is but a small fraction of the size of the smallest input.

The contribution of DE in this instance is twofold. First is that the problem formulation is quite easy. This saves on programming time. More importantly, perhaps, is that run time is also kept within reason, in this case around two minutes. When an all out effort using dedicated integer linear programming (ILP) technology was attempted, it simply hung (it did not run to completion in 24 hours). One could perhaps alter such code to return a result once it is deemed "good enough," but that involves intervention in the depths of what is already complicated ILP code. By contrast, DE offers complexity that is well understood. Under the quite reasonable assumption that every fitness evaluation is roughly comparable in computational cost, call it f, if we run for g generations and use c chromosomes, the expected run time will simply be $O(gcf)$.

5 Matrix column sum minimization

We are given a sparse matrix comprised of, say, symbols. Each symbol can appear in multiple rows, but appears only once in a given row. We are required to select exactly one symbol per row. The object is to do so in such a way that we minimize the number of repeats. Moreover, we want to penalize a triple of repeats more than two doubles, etc. The idea is to have the selected symbols be "as unique as possible" (to abuse language ever so slightly).

This is an exemplar of a problem that arose in house. The goal is to obtain a good set of disambiguating terms for a grammar intended for handling certain natural language queries. Matrix rows become grammar "rules". Columns are the set of grammar "primitives," which might be phrase types or actual words. The task then is to select one primitive to represent each rule, and try to minimize repeats so that the representation is close to unique; this allows for faster look-up of rules by their associated primitive. That is to say, we seek a set of primitives to act as classifiers for queries of look-up rules.

We will denote symbol k in row j simply by $b(j,k)$. In our example problem, each row will contain 3 symbols, though in general it is not required that the number of symbols per row be constant. Below is a very small example to illustrate the problem formulation:

$$\begin{pmatrix} b(1,2) & b(1,4) & b(1,8)\\ b(2,6) & b(2,4) & b(2,5)\\ b(3,6) & b(3,7) & b(3,4)\\ b(4,4) & b(4,2) & b(4,7)\end{pmatrix}.$$

To explain our meaning, the first row is comprised of symbols 2, 4, and 8, the second row uses symbols 6, 4, and 5, etc. We now formulate our optimization problem. We will insist that all variables lie between 0 and 1, be integers (so of course it is a 0-1 problem), and that each row sum be one (so there is exactly one 1 per row). We will minimize the sum of fourth powers of the column sums. Any power larger than one might be reasonable, but in using fourth powers we seem to do a good job of forcing (near) uniqueness of symbol selections in columns. This useful ploy is of course not amenable to an ILP formulation. For the toy example matrix above, our variables are simply the matrix elements, which

are constrained to take on values of 0 or 1. Our objective function would be the following:

$$b[1,8]^4 + b[2,5]^4 + (b[2,6] + b[3,6])^4 + (b[1,2] + b[4,2])^4 + (b[1,4] + b[2,4] + b[3,4] + b[4,4])^4 + (b[3,7] + b[4,7])^4.$$

Notice that since the value 4 appears in every row, the objective function has a term of the form $(b[1,4] + b[2,4] + b[3,4] + b[4,4])^4$. Thus if we use that value three times, say, the contribution from that term will be 3^4, or 81. Likewise we have some values that appear in pairs of rows, thus allowing for contributions of 16 should both be used. This may give some insight into why this objective function is well suited to the purpose at hand.

Our full problem is a random example where we have 60 rows, and we use 50 symbols. It is set up in the code appendix, once again using but little code. The actual run took about three minutes.

Now let us check the quality of the result. The list below shows how many times each symbol was used. What we hope to see are no values larger than 2, and as few of those as possible.

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

Observe that we used 11 symbols twice and no symbol more than that; all others appear at most once. Given the way we had set up this random problem instance, the best possible outcome would have to use at least 10 symbols twice (we have to make 60 choices from a set of 50 values). So clearly we are either optimal, or quite close, for this particular instance. We again point out that we are well served by an objective function that is nonlinear. While it is readily handled by DE, it may not be so nice for more classical methods, given the further restriction that variables be integers.

The random problem we created for this example is in important respects different from what one might expect to encounter in the intended application. Since words are not used with equal frequencies in common language, we expect clustering (which makes the goal more difficult). And some words or phrases might indeed by quite uncommon, hence making the task easier in places. It remains to be seen how well this approach will perform on practical problems. All the same, the example gives some indication that one might obtain a good set of classifying primitives in reasonable time. For example, when I doubled the problem size parameters from (50,60) to (100,120), I obtained a result with a modest number of pair overlaps, but no triples, in about 15 minutes. We would expect speed to degrade, but not forbiddingly so, on problems of, say, ten times as many symbols and rows. The objective function evaluations would scale roughly linearly. We would, however, expect also to need more generations and more chromosomes to achieve reasonable results. This could lead to one or two more orders of magnitude in algorithmic complexity, which would amount to a run time of anywhere from a few hours to a few days. This is, for many optimization purposes, not so bad; one time (or "a few times") computations are by no means uncommon in practical settings.

6 That elusive `"CrossProbability"` option

The diligent reader has checked the Code appendix and has noticed that we use an option setting `"CrossProbability" -->9/10` (the default value is 1/2). This calls for explanation. First we describe what this value controls. In differential evolution one constructs, for each chromosome, a child chromosome against which the parent will compete, with the better one moving to the next generation. To create a child, a trial chromosome is first constructed according to a certain method, the details of which are discussed in several of the references. The child is then formed from this trial chromosome and the parent by using random values between 0 and 1 for each position in the chromosome. If a value is larger than the `"CrossProbability"` setting we take that element from the trial chromosome, otherwise we take it from the corresponding position in the parent. I note in passing that this is the opposite of what is most commonly done in the literature. (How did that come about? I will surmise that we got it backwards a long time ago, and never noticed the discrepancy.) I also will note that the above brief explanation is lacking in a few details (unimportant for our present purposes), and also does not do justice to the variety of different crossover methods explored in several of the references.

We now consider the issue of how best to set this value. For many optimization tasks, especially in the realm of continuous functions, settings near 0.5 seem to work well (some references discuss limitations in this regard). Moreover, the quality of result is often not terribly sensitive to this setting. When tackling combinatorial optimization problems it is, alas, a very different matter. An emerging view, as indicated for example in [13,27], is that certain types of such problem are better handled by retaining mostly genes from the parent chromosome, and other problem types do well with exactly the opposite strategy. We should mention here that by problem type we really refer to the implementation details of how the objective function is formed. That is to say, it is genotype rather than phenotype that matters for this purpose. Indeed, the same underlying problem might have different formulations that lead to opposite optimal settings for the `"CrossProbability"`. For some examples of this phenomenon, see [8].

The main feature in determining which of these strategies is likely to work better is referred to as the separability of the objective function. This is a measure of the extent to

which the objective depends on interdependencies between parameters rather than the specifics of those parameter values themselves. More specifically, a problem is termed separable if the optimal value of any parameter is independent of the values of the remaining parameters; this is a situation of parameters having low interdependency.

For example, a fairly common specification of an objective function in a combinatorial setting involves using the values to determine an ordering of some set of values (sometimes referred to as a "rank based" or "relative position indexing" approach). In this setting a change in a small number of parameters can drastically alter the effect of the remaining parameter values in terms of evaluating the objective function. A problem specification of this type is nonseparable. In the opposite direction, one might use parameter values as a means to select an ordering for combinatorial problems using a shuffle approach [8, 23]. In this setting the values in different positions on a chromosome act in a way that is moderately, though not entirely, independent of one another. This is thus much closer to being a separable problem formulation.

Experiments indicate that nonseparable problems do better when parent values are retained in child chromosomes with high probability [8,13,27]. We remark however, that some nonseparable problem formulations in [8] seem to work better with settings that favor using more values from the trial chromosome than the parent (i.e., low settings for `"CrossProbability"`). These opposite findings justify the use of "elusive" in the above section title. That said, experiments in [8,13,27] indicate that in many cases, whether separable or nonseparable, either extreme of cross probability settings will tend to give better results than values near the middle of the unit interval. Moreover [27] shows that it can be beneficial to use both types of setting in the same problem; this can be controlled by random choices at each step.

Finally we remark that this brief discussion hardly does justice to the topic of cross probability tuning. The references noted above contain a wealth of information in this regard.

7 Scheduling concurrent jobs for several machines

We now turn to a problem that also arose in house. One department needs to run certain programs nightly. They have access to a fixed number of machines, and have good approximations for the run time of each program. We will model the simple case of homogeneous machines, so each job time will be independent of machine used. These times are only approximate, and moreover we want a schedule that is robust (in the sense that a failure of some assumption such as program time will not do too much damage). Typically one might augment them by a modest percentage; we will simply assume that has already been done, and take our set of job times as given.

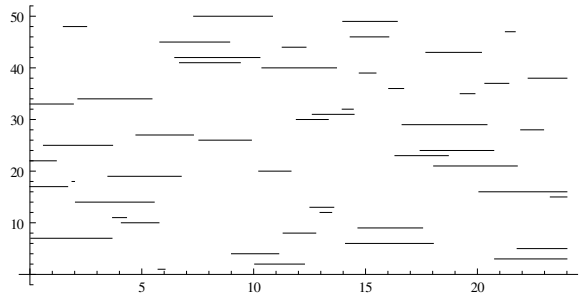

Figure 1: Job times.

Our variables will be the starting times for each job. Our total time period will be one day (consider the situation where the set of jobs must be run daily). An obvious constraints is that all jobs start at a nonnegative time and that they start more than their length prior to the 24 hour limit. Moreover we are not allowed to have more jobs than available machines at any one time. To check this we must test, at each starting time, that there is at least one machine free. This is coded as follows. We have a summation constraint for every job variable. The sum is over all other jobs. Each contributes a value of one precisely if it starts before, and ends after, the start time of the one under consideration, else zero. It is these sums, one such for each job, that must be at least one less than the total number of machines.

We will take the view that we want to discourage too many from running at one time by having an objective function that contributes in increasing amounts when the number of concurrent jobs exceeds some threshold (this acts in such a way as to form gaps between jobs, which can be good in situations where some jobs run a bit long). For this purpose a plausible value is two less than maximum allowed at any time. The function we used to this end sums the cubes of the number of jobs, less two, that are running at the time each new one begins.

The code appendix has an example worked out. We generated 50 jobs of random time lengths from 0 to 4 hours. We allow five machines and have a full 24 hour period. The total time required for our random example is around 103.6 hours. Allowing for use of five machines, it is easy to see that scheduling these is by no means trivial. The optimization call (see the appendix for full encoding) specified 250 chromosomes and up to 2000 generations. The run time was slightly over one hour.

All constraints were met by the eventual solution. The graph in Figure 1 indicates when each job is active, as scheduled using the methodology described above. Time is the horizontal axis. The vertical axis simply records which of the 50 jobs is being shown at that height. In simple geometric terms, the number of active jobs at a given time is found by slicing with a vertical line and counting the number of horizontal bars hit.

This example has a mix of components that make it challenging. One is that a local optimizer will have trouble avoiding bad local optima. Another is that it mixes discrete active job counts with continuous starting times. Yet another is that the objective function itself has sharp jumps when active job counts increase (beyond a threshold). Moreover we really seem to need a nonlinear rise, which rules out use of integer linear programming for this formulation of the problem. Given the fact that we are able to get a viable result, using DE, in an hour or so, we will claim that DE has served well in this situation.

8 Conclusions and future directions

We have used the method of differential evolution, with measurable success, on several challenging discrete optimization problems from diverse application areas. The common thread is that none were obvious candidates for better known methods (or, in the knapsack example, the superior method simply did not work out). In each case we obtained useful results. This was done at minimal cost in both programming and computational resources; all required but a few lines of code and run times of no more than a few minutes. While the objective function formulations were in some cases nontrivial, we think that the present work gives useful ideas for how one might approach similar problems.

There are many open questions and considerable room for further development. Here are a few possible directions.

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.

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 for the sort of problems we showed.

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 place, so that improvements are passed along to subsequent generations. Implementation in *Mathematica* is illustrated in some detail in chapter four of [10].

Some ideas related to item 1 are discussed in [27]. Aspects of 2 are explored in [6]. Issues of self-adaptive tuning of differential evolution are discussed in some detail in [1,16,19,24,26]. An exposition of early efforts along these lines, for genetic algorithms, appears in [5].

9 Code appendix

9.1 Invoking differential evolution within *Mathematica*

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

```
NMinimize[{objective,constraints},variables,
 Method-->{"DifferentialEvolution",methodopts},
 otheropts]
```

Here `methodopts` might include setting to non-default values any or all of the options indicated below. We will show usage of some in the examples. 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. Further details may be found in the program documentation, all of which is available online [25]. Moreover numerous implementation details are discussed in [2].

9.2 Chemical equation of state

$$
\begin{aligned}
\text{data} = \{&\{0.492836,-2.76349979\},\{0.484912,-2.71627714\},\\
&\{0.476987,-2.66934661\},\{0.461139,-2.57627064\},\\
&\{0.44529,-2.48420922\},\{0.429441,-2.39306352\},\\
&\{0.405668,-2.2578664\},\{0.381894,-2.12420539\},\\
&\{0.358121,-1.99180889\},\{0.342272,-1.90410278\},\\
&\{0.318499,-1.77321817\},\{0.318499,-1.77320734\},\\
&\{0.286801,-1.59966342\},\{0.286801,-1.59964561\},\\
&\{0.263028,-1.46992327\},\{0.263028,-1.46990573\},\\
&\{0.239254,-1.34031315\},\{0.239254,-1.34031185\},\\
&\{0.223406,-1.25390787\},\{0.199632,-1.12418091\},\\
&\{0.183783,-1.03751393\},\{0.167935,-0.95065758\},\\
&\{0.144161,-0.81990054\},\{0.128312,-0.73235589\},\\
&\{0.128312,-0.7323321\},\{0.112464,-0.64440847\},\\
&\{0.096615,-0.55600443\},\{0.08869,-0.51160367\},\\
&\{0.080766,-0.46706783\},\{0.072841,-0.42236874\},\\
&\{0.064917,-0.37750794\},\{0.056993,-0.33245946\},\\
&\{0.049068,-0.28720197\},\{0.041144,-0.24171452\},\\
&\{0.033219,-0.19597321\},\{0.029257,-0.17298665\},\\
&\{0.025295,-0.14992861\},\{0.025295,-0.14990853\},\\
&\{0.021333,-0.12677606\},\{0.01737,-0.10352715\},\\
&\{0.014201,-0.08485674\},\{0.011823,-0.07080379\},\\
&\{0.009446,-0.05670636\}\};
\end{aligned}
$$

```
data2=Map[{{#[[1]]},#[[2]]}&,data];
findFewnomial[data_,vars_,nterms_,iters_,npts_,den_:1,
     prnt_:False,lvars_:{c,j}]:=
   Catch[Module[
     {coeffs,expons,nvars=Length[vars],poly,err,
     cminfunc,expvals,min1,min2,coeffvals},
     coeffs=Array[lvars[[1]],nterms];
     expons=Array[lvars[[2]],{nvars,nterms}];
     poly=coeffs.Times@@(vars^(expons/den));
     expons=Flatten[expons];
     err=Total[Map[(poly/.Thread[vars-->#[[1]]])
                  -#[[2]]&,data]^2];
     If[prnt,Print["polynomial is "];Print[poly];
        Print["sum of squared discrepancies is "];
        Print[err]];
     cminfunc[obj_,cvars_,evars_,evals:{_Integer..}]:=
        First[Quiet[FindMinimum[obj/.evars-->evals,
              cvars]]];
     expvals=expons;
     min1=NMinimize[{cminfunc[err,coeffs,expons,
                              expvals],
        Flatten[{Element[expvals,Integers],
                 Map[0<=#&,expvals]}]},
        expvals,MaxIterations-->iters];
        Method-->{"DifferentialEvolution",
        "SearchPoints"-->npts},
        If[Head[min1]===NMinimize||!
                        FreeQ[min1, Indeterminate],
           Throw[$Failed],
           {min1,expvals}=min1];
     {min2,coeffvals}=FindMinimum[err/.expvals,coeffs];
     {min2,poly/.Join[expvals,coeffvals]}]
     ]
```

Simple example:

```
Timing[findFewnomial[Take[data2,4],{x},2,20,5,6,True]]
```

Actual problem:

```
Timing[findFewnomial[data2,{x},4,200,50,6]]
```

9.3 File storage

bigvals = {1305892864, 1385113088, 856397968,
1106152425, 1647145093, 1309917696, 1096825032,
1179242496, 1347631104, 696451130, 746787826,
1080588288, 1165223499, 1181095818, 749898444,
1147613713, 1280205208, 1242816512, 1189588992,
1232630196, 1291995024, 911702020, 1678225920,
1252273456, 934001123, 863237392, 1358666176,
1714134790, 1131848814, 1399329280, 1006665732,
1198348288, 1090000441, 716904448, 677744640,
1067359748, 1646347388, 1266026326, 1401106432,
1310275584, 1093615634, 1371899904, 736188416,
1421438976, 1385125391, 1324463502, 1489042122,
1178813212, 1239236096, 1258202316, 1364644352,
557194146, 555102962, 1383525888, 710164700,
997808128, 1447622656, 1202085740, 694063104,
1753882504, 1408100352};

targ = 8547993600;

```
closestSum[vals_,target_,maxiters_:100,
     sp_:Automatic,cp_:.5,prnt_:False]:=
   Module[{vars,x,len,obj,constraints,max,best},
    len=Length[vals];
    vars=Array[x,len];
    Format[x]:=x;
    Format[x,TraditionalForm]:=x;
    obj=vars.vals;
    constraints=Join[Map[0<=#<=1&,vars],
      {obj<=target,Element[vars,Integers]}];
    If[prnt,
       Print["objective: ",obj];
       Print["constraints: ",TableForm[constraints]]];
    {max,best}=
      NMaximize[{obj,constraints},vars,
       MaxIterations-->maxiters,
       Method-->{"DifferentialEvolution",
       "SearchPoints"-->sp,"CrossProbability"-->cp}];
    {max,target-max,N[(target-max)/target],vars/.best}]
```

Simple example:

```
SeedRandom[11111];
Timing[Quiet[closestSum[RandomInteger[{1000},5],
                        2000,80,20,True]]]
```

Actual problem:

```
Timing[Quiet[closestSum[bigvals,targ,300,100]]]
```

9.4 Column sum minimization

First we set up a random problem of reasonable size.

```
n=50;
SeedRandom[1111];
mat=Table[RandomSample[Range[n],3],{60}];
rowlists=MapIndexed[b[#2[[1]],#]&,mat,{2}];
collists=GatherBy[Flatten[rowlists],#[[2]]&];
TableForm[Take[rowlists,4]]
```

Our objective will use fourth powers of column sums. Unique individuals contribute values of 1, pair collisions 2, triple collisions 3, etc. (and these values are raised to the fourth power).

```
vars=Flatten[rowlists];
c0=Map[0<=#<=1&,vars];
c1={Element[vars,Integers]};
c2=Map[Total[#]==1&,rowlists];
constraints=Join[c0,c1,c2];
obj=Total[Map[Total[#]^4&,collists]]
```

We run the example:

```
Timing[Quiet[{min,best}=
  NMinimize[{obj,constraints},vars,MaxIterations-->500,
   Method-->{"DifferentialEvolution",
   "CrossProbability"-->9/10,"SearchPoints"-->50}];]]
{175.735,Null}
```

We check the quality of the result by looking at how many pairs, triples, etc. appear in it.

```
colsums=Total[Array[b,{60,50}]/.best/.b[_,_]:-->0]
Count[colsums,2]
{1,1,1,1,2,1,0,1,1,1,2,1,1,1,2,1,1,1,1,2,1,1,1,1,1,
 2,1,1,2,2,2,1,1,1,1,1,1,1,2,1,2,1,1,1,1,2,1,1,1,1}
11
```

So there are no collisions beyond pairs, and only 11 of those.

9.5 Job scheduling

First we set up a random example. It is similar in size and scope to the actual problem that walked into my office.

```
SeedRandom[1111];
numjobs=50;
daylength=24;
joblengths=RandomReal[{0,4},numjobs];
Total[joblengths]
103.614
```

The problem is set up as follows:

```
machines=5;
starttimes=Array[t,numjobs];
endtimes=starttimes+joblengths;
sdiff[j_,k_]:=starttimes[[j]]-starttimes[[k]];
ediff[j_,k_]:=endtimes[[j]]-endtimes[[k]]
```

Our variables are the starting times for each job. Constraints and objective function are as described above.

```
overlap[j_]:=
Sum[UnitStep[sdiff[j,k]]*UnitStep[sdiff[k,j]
            +joblengths[[k]]],{k,1,j-1}]+
Sum[UnitStep[sdiff[j,k]]*UnitStep[sdiff[k,j]
            +joblengths[[k]]],{k,j+1,numjobs}]
c1=Thread[0<=starttimes<=daylength-joblengths];
c2=Table[overlap[j]<=machines-1,{j,numjobs}];
constraints=Join[c1,c2]
```

We now do our optimization:

```
Timing[
{min,vals}=NMinimize[{weightedoverlaps,constraints},
                     starttimes,
  Method-->{"DifferentialEvolution",
            "SearchPoints"-->250,"CrossProbability"-->.9,
    "PostProcess"-->False},MaxIterations-->2000];]
```

The code below was used to form the picture of our result:

```
lines=MapIndexed[Line[{{#1[[1]],#2[[1]]},{#1[[2]],
                       #2[[1]]}}]&,
Transpose[{starttimes,starttimes+joblengths}/.vals]];
Graphics[lines,Axes-->True,AxesOrigin-->{0,0},
         AspectRatio-->1/2]
```

An ILP approach, which is quite fast, is as follows. We have a variable for each combination of machine and job. For each such $m_{i,j}$ we assign the value of 1 if machine i is to run job j, else 0. We have constraints that all variables are in the range $(0, 1)$, all are integer valued, and that the total times incurred by each machine not exceed 24. For balance and also to allow a bit of leeway we actually constrain total times on each machine to lie between 20 and 23 hours. As we only need satisfy constraints there really is no need for an objective function.

```
SeedRandom[1111];
numjobs=50;
daylength=24;
joblengths=RandomReal[{0,4},numjobs];
machines=5;
vars=Array[m,{machines,numjobs}];
fvars=Flatten[vars];
c1=Map[0<=#<=1&,fvars];
c2=Thread[Total[vars]==1];
c3=Thread[20<=vars.joblengths<=23];
constraints=Join[c1,c2,c3];
{min,vals}=Minimize[{1,constraints},fvars,Integers]
```

We check that each machine is occupied for less than 24 hours.

```
vars.joblengths/.vals {21.5142,21.7549,20.1749,
                       20.0772,20.0923}
```

This shows which jobs, listed by number, get run on which machine.

```
TableForm[Map[Last,Split[Position[vars/.vals,1],
                         #1[[1]]===#2[[1]]&],{2}]]
1  11 12 14 18 26 32 35 39 41 42 44 46 47 49
2  22 25 33 34 37 40 43 45
3  4  6  13 19 29 38 48
5  7  8  9  10 15 16 50
17 20 21 23 24 27 28 30 31 36
```

This method works quite well. Nevertheless, we believe that the DE approach offers considerable flexibility in situations where there might be nonlinear constraints on the scheduling, or perhaps disjunctions of linear constraints (e.g., "jobs j and k must run consecutively on the same machine").

Acknowledgments I thank the anonymous referees of this and a prior draft for providing several useful remarks and references. I also thank Kenneth Price, Rainer Storn, and Ivan Zelinka for helpful discussions via e-mail on related aspects regarding implementation and application differential evolution. Finally I am happy to acknowledge very helpful comments from Daniela Zaharie regarding separable vs. inseparable problems, following a talk I gave on this topic at SYNASC 2010. Her remarks served as motivation for the `"CrossProbability"` discussion in the present work, and some of her published work also influenced that section.

References

[1] J. Brest, S. Greiner, B. Bošković, M. Mernik, and V. Žumer, *Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems*, IEEE Transactions on Evolutionary Computation, 10 (2006), 646–657.

[2] B. Champion, *Numerical optimization in Mathematica: an insider's view of NMinimize*, in Proc. of the 6th World Conference on Systemics, Cybernetics, and Informatics, N. Callaos, T. Ebisuzaki, B. Starr, J. M. Abe, and D. Lichtblau, eds., vol. 16, Orlando, FL, 2002, 136–140.

[3] E. da Silva, H. Barbosa, and A. Lemonge, *An adaptive constraint handling technique for differential evolution with dynamic use of variants in engineering optimization*, Optimization and Engineering, 12 (2011), 31–54.

[4] K. Gisvold and J. Moe, *A method for nonlinear mixed-integer programming and its application to design problems*, Journal of Engineering for Industry, 94 (1972), 353–364.

[5] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing, Boston, MA, 1989.

[6] C. Jacob, *Illustrating Evolutionary Computation with Mathematica*, Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[7] N. Krasnogor and J. Smith, *A tutorial for competent memetic algorithms: model, taxonomy, and design issues*, IEEE Transactions on Evolutionary Computation, 9 (2005), 474–488.

[8] D. Lichtblau, *Discrete optimization using Mathematica*, in Proc. of the 6th World Conference on Systemics, Cybernetics, and Informatics, N. Callaos, T. Ebisuzaki, B. Starr, J. M. Abe, and D. Lichtblau, eds., vol. 16, Orlando, FL, 2002, 169–174.

[9] D. Lichtblau, *Making change and finding repfigits: balancing a knapsack*, in Proc. of the 2nd International Conference on Mathematical Software (ICMS'06), A. Iglesias and N. Takayama, eds., Castro-Urdiales, Spain, 2006, 182–193.

[10] D. Lichtblau, *Relative position indexing approach*, in Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization, G. C. Onwubolu and D. Davendra, eds., Springer-Verlag, Berlin, 2009, 81–120.

[11] R. Lougee-Heimer, *The Common Optimization INterface for Operations Research: promoting open-source software in the operations research community*, IBM Journal of Research and Development, 47 (2003), 57–66.

[12] G. Mitchell, D. O'Donoghue, D. Barnes, and M. McCarville, *GeneRepair: a repair operator for genetic algorithms*, in Proc. of the Genetic and Evolutionary Computation Conference, Chicago, IL, 2003, 88–93.

[13] J. Montgomery and S. Chen, *An analysis of the operation of differential evolution at high and low crossover rates*, in Proc. of the IEEE Congress on Evolutionary Computation (CEC '10), Barcelona, Spain, 2010, 1–8.

[14] P. Moscato, *On evolution, search, optimization, genetic algorithms, and martial arts*, Concurrent computation program 826, California Institute of Technology, 1989.

[15] P. Nowak, R. Kleinrahm, and W. Wagner, *Measurement and correlation of the (p,ρ,T) relation of nitrogen. II. Saturated-liquid and saturated-vapour densities and vapour pressures along the entire coexistence curve*, Journal of Chemical Thermodynamics, 29 (1997), 1157–1174.

[16] M. G. Omran, A. Salman, and A. P. Engelbrecht, *Self-adaptive differential evolution*, in Proc. of the International Conference on Computational Intelligence and Security (CIS'05), Xian, China, 2005, 192–199.

[17] G. Pampara, A. P. Engelbrecht, and N. Franken, *Binary differential evolution*, in Proc. of the IEEE Congress on Evolutionary Computation (CEC '06), Vancouver, BC, 2006, 1873–1879.

[18] K. Price, R. Storn, and J. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*, Natural Computing Series, Springer-Verlag New York, Secaucus, NJ, 2005.

[19] A. K. Qin, V. L. Huang, and P. N. Suganthan, *Differential evolution algorithm with strategy adaptation for global numerical optimization*, IEEE Transactions on Evolutionary Computation, 13 (2009), 398–417.

[20] U. Setzmann and W. Wagner, *A new method for optimizing the structure of thermodynamic correlation equations*, International Journal of Thermophysics, 10 (1989), 1103–1126.

[21] P. Sisak, *0/1 knapsack-like minimalization problem and file system access*. http://forums.wolfram.com/mathgroup/archive/2010/Mar/msg00905.html, 2010.

[22] R. Storn and K. Price, *Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces*, Journal of Global Optimization, 11 (1997), 341–359.

[23] D. M. Tate and A. E. Smith, *A genetic approach to the quadratic assignment problem*, Computers and Operations Research, 22 (1995), 73–83.

[24] J. Tvrdík, *Adaptation in differential evolution: a numerical comparison*, Applied Soft Computing, 9 (2009), 1149–1155.

[25] Wolfram Research, *Mathematica 8*. http://reference.wolfram.com/mathematica/ref/NMinimize.html, 2010.

[26] D. Zaharie, *Control of population diversity and adaptation in differential evolution algorithms*, in Proc. of the 9th International Conference on Soft Computing, R. Matoušek and P. Ošmera, eds., Brno, Czech Republic, 2003, 41–46.

[27] D. Zaharie, *Influence of crossover on the behavior of differential evolution algorithms*, Applied Soft Computing, 9 (2009), 1126–1138.