Chapter 7

Ordered Structures

©2003 by Dan Ashlock

The data structures we have evolved thus far have all been arrays or vectors of similar elements, be they characters, real numbers, the ships's systems from Sunburn, or states of a finite state automaton. The value at one state in a gene has no effect on what values may be present at another location, except for non-explicit constraints implied by the fitness function.

In this chapter, we will work with lists of items called *permutations* in which the list contains a specified collection of items once each. We will store the permutations as lists of integers $1, 2, ..., n$ varying only the order in which the integers appear. Genes of this type are used for well-known problems such as the *Traveling Salesman* problem. Just as we used the simple string evolver in Chapter 2 to learn how evolutionary algorithms worked, we will start with easy problems to learn how systems for evolving ordered genes work. We will also look at applications of permutations to highly technical mathematical problems.

The basic definition of a permutation is simple: an order in which to list a collection of items, no two of which are the same. To work with structures of this type, we will need a bit of algebra and a cloud of definitions.

Definition 7.1 *A* **permutation** *of the set* $N = \{0, 1, \ldots, n-1\}$ *is a bijection of* N *with itself.*

Theorem 7.1 *There are* $n! := n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$ *different permutations of* n *items.*

Proof:

Order the n items. There are n choices of items onto which the first item may be mapped. Since a permutation is a bijection there are $n-1$ items onto which the second item may be mapped. Continuing in like fashion, we see the number of choices of destination for the ith item is (n-i+1). Since these choices are made independently of one another with past choice

not influencing present choice among the available items, the choices multiply - yielding the stated number of permutations. □

Example 7.1 *There are several ways to represent a permutation. Suppose the permutation f is: f(0)=0, f(1)=2, f(2)=4, f(3)=1, and f(4)=3. It can be represented in* **two-line** *notation:*

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 1 & 3 \end{pmatrix}$$

Two line notation lists the set in "standard" order in its first line and in the permuted order in the second line. **One line** *notation:*

$$\begin{pmatrix} 0 & 2 & 4 & 1 & 3 \end{pmatrix}$$

is two-line notation with the first line gone. Another notation commonly used is called **cycle** *notation. Cycle notation gives permutations as a list of disjoint cycles, ordered by their leading items, with each cycle tracing how a group of points are taken to one another. The cycle notation for our example is:*

$$(0)(1\ 2\ 4\ 3),$$

because 0 goes to 0, 1 goes to 2 goes to 4 goes to 3 returns to 1.

Be careful! If the items in a permutation make a single cycle, then it is easy to confuse one-line and cycle notation.

Example 7.2 *Here is a permutation of the set* $\{0, 1, 2, 3, 4, 5, 6, 7\}$ *shown in one-line, two-line, and cycle notation.*

Two line:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 3 & 4 & 7 & 5 & 6 & 0 & 1 \end{pmatrix}$$

One line:

$$\begin{pmatrix} 2 & 3 & 4 & 7 & 5 & 6 & 0 & 1 \end{pmatrix}$$

Cycle:

$$(0\ 2\ 4\ 5\ 6)(1\ 3\ 7)$$

A permutation uses each item in the set once. The only real content of the permutation is the order of the list of items. Since permutations are functions, they can be composed.

Definition 7.2 Multiplication *of permutations is done by composing them.*

$$(f * g)(x) := f(g(x)) \tag{7.1}$$

Definition 7.3 *The permutation that takes every point to itself is the* **identity** *permutation. We give it the name e.*

Since permutations are bijections, it is possible to undo them and so permutations have inverses.

Definition 7.4 *The* **inverse** *of a permutation $f(x)$ is the permutation $f^{-1}(x)$ such that*

$$f(f^{-1}(x)) = f^{-1}(f(x)) = x.$$

In terms of the multiplication operation the above would be written

$$f * f^{-1} = f^{-1} * f = e.$$

Example 7.3 *Suppose we have the permutations in cycle notation $f = (02413)$ and $g = (012)(34)$. Then:*

$$f * g = f(g(x)) = (0314)(2),$$
$$g * f = g(f(x)) = (0)(1423),$$
$$f * f = f(f(x)) = (04321),$$
$$g * g = g(g(x)) = (021)(3)(4),$$
$$f^{-1} = f^{-1}(x) = (03142), \text{ and}$$
$$g^{-1} = g^{-1}(x) = (021)(34).$$

-

Cycle notation may seem sort of weird at first, but it is quite useful. The following definition and theorem will help you to see why.

Definition 7.5 *The* **order** *of a permutation is the smallest number k such that if the permutation is composed with itself k times, the result is the identity permutation e. The order of the identity is 1, and all permutations of a finite set have finite order.*

Theorem 7.2 *The order of a permutation is the least common multiple of the lengths of its cycles in cycle notation.*

Proof:

Consider a single cycle. If we repeat the action of the cycle a number of times less than its length, then its first item is taken to some other member of the cycle. If the number of repetitions is a multiple of the length of the cycle, then each item returns to its original position. It follows that, for a permutation, the order of the entire permutation is a common multiple of its cycle lengths. As the action of the cycles on their constituent points is independent, it follows that the order is the least common multiple. □

Definition 7.6 *The* **cycle type** *of a permutation is a list of the lengths of the permutation's cycles in cycle notation. The cycle type is an unordered partition of n into positive pieces.*

Example 7.4 *If $n = 5$, then the cycle type of e (the identity) is 1 1 1 1 1. The cycle type of (0 1 2 3 4) is 5. The cycle type of (0 1 2)(3 4) is 3 2.*

n	Max Order	n	Max Order	n	Max Order
1	1	13	60	25	1260
2	2	14	84	26	1260
3	3	15	105	27	1540
4	4	16	140	28	2310
5	6	17	210	29	2520
6	6	18	210	30	4620
7	12	19	420	31	4620
8	15	20	420	32	5460
9	20	21	420	33	5460
10	30	22	420	34	9240
11	30	23	840	35	9240
12	60	24	840	36	13,860

Table 7.1: The maximum order of a permutation of n items

Table 7.1 gives the maximum order possible for a permutation of n items ($n \leq 36$). The behavior of this number is sort of weird, growing abruptly with n sometimes and staying level other times. More values for the maximum order of a permutation of n items may be found in the *On Line Encyclopedia of Integer Sequences*[32].

Example 7.5 *Here are some permutations in cycle notation and their orders.*

Permutation	*Order*
(0 1 2)(3 4)	*6*
(0 1)(2 3 4 5)(6 7)	*4*
(0 1)(2 3 4)(5 6 7 8 9)	*30*
(0 1 2 3 4 5 6)	*7*

Definition 7.7 *A* **reversal** *in a permutation is any pair of items such that, in one-line notation, the larger item comes before the smaller item.*

Example 7.6 *Here are some permutations in one-line notation and their number of reversals.*

Permutation	*Order*
1 2 3 4 5 6 7 8 9	*0*
2 1 0 3 4 6 5 8 7	*5*
1 6 0 5 8 4 3 7 2	*17*
8 6 5 4 2 7 3 1 0	*31*
9 8 7 6 5 4 3 2 1	*36*

Theorem 7.3 *The maximum number of reversals of a permutation of n items is*

$$\frac{n(n-1)}{2}.$$

The minimum number is zero.

Proof:

It is impossible to have more reversals than the number obtained when larger numbers strictly precede smaller ones. In that case, the number of reversals is the number of pairs (a, b) of numbers with a larger than b. There are $\binom{n}{2}$ such pairs, yielding the formula desired. The identity permutation has no reversals, yielding the desired lower bound. □

Reversals and orders of permutations are easy to understand and simple to compute. Maximizing the number of reversals and maximizing the order of a permutation are simple problems we will use to dip our toes into the process of evolving permutations.

Definition 7.8 *A* **transposition** *is a permutation that exchanges two items and leaves all others fixed. E.g.: (1 2)(3)(4) - cycle notation.*

Theorem 7.4 *Any permutation on n items can be transformed into any other permutation on n items by applying* at most $n-1$ *transpositions.*

Proof:

Examine the first $n-1$ places of the target permutation. Transpose these items into place one at a time. By elimination, the remaining item is also correct. □

7.1 Evolving Permutations

In order to evolve permutations, we will have to have some way to store them in a computer. We will experiment with more than one way to represent them. Our first will be very close to one-line notation.

Definition 7.9 *An array containing a permutation in one-line notation is called the* **standard representation**.

While the standard representation for permutations might seem quite natural, it has a clear flaw. If we fill in the data structure with random numbers, even ones in the correct range, it is easy to get a non-permutation. This means we must be careful how we fill in the array and how we implement the variation operators.

A typical evolutionary algorithm generates a random initial population. Generating a random permutation is a bit trickier than filling in random characters in a string, because we have to worry about having each list item appear exactly once. The code given in Figure 7.1 can be used to generate random permutations.

```
CreateRandomPerm(perm p,int n){//assume p is an array of n integers

int i,rp,sw;  //loop index, random position, swap

  for(i=0;i<n;i++)perm[i]=i;    //fill in the identity permutation

  for(i=0;i<n-1;i++){//for all entries
    rp=random(n-i)+i;  //get a random number in the range i..(n-1)

    sw=p[i];           //swap the
    p[i]=p[rp];        //current
    p[rp]=sw;          //and random entry

  }

}
```

Figure 7.1: Code for creating a random permutation of the list 0,1,...,(n-1) of integers

Now that we have a way of generating random permutations, we need to choose the variation operators. The choice of variation operators has a substantial impact on the performance of an evolutionary algorithm. Mutation is easy; we will simply use transpositions.

Definition 7.10 *A* **transposition mutation** *is the application of a transposition to a permutation. An n-transposition mutation is the application of n transpositions to a permutation.*

Crossover is a more challenging operation than mutation. If we try to apply our standard crossover operator to permutations, there is a high probability of destroying the property that each item appears only once in the list. The following is a standard crossover operation for permutations.

Definition 7.11 *A* **one point partial preservation** *crossover of two permutations in the standard representation is performed as follows. A single crossover point is chosen. In each permutation, those items present at or before the crossover point are left untouched. Those items after the crossover point also appear after the crossover point, but in the order that they appear in the other permutation. This crossover operator produces two permutations from two permutations but is fairly destructive.*

Example 7.7 *Let's look at some examples of one point partial preservation crossover. All the permutations are given in one-line notation.*

Parents	*Crossover Point*	*Children*
(0 1 2 3 4) *(1 4 3 2 0)*	*3*	*(0 1 2 4 3)* *(1 4 3 0 2)*
(0 3 5 4 7 6 1 2) *(7 3 4 1 2 0 6 5)*	*4*	*(0 3 5 4 7 1 2 6)* *(7 3 4 1 0 5 6 2)*
(0 1 2 3 4 5) *(1 2 3 4 5 0)*	*1*	*(0 1 2 3 4 5)* *(1 0 2 3 4 5)*

We now have sufficient machinery to perform a simple experiment with evolving permutations. We will be looking for permutations with a large number of reversals.

Experiment 7.1 *Either write or obtain code that can randomly generate permutations and can apply transposition mutations and one point partial preservation crossover. Use the standard representation for permutations. Also obtain or write code for computing the number of reversals in a permutation. Using a population of 120 permutations write a steady state evolutionary algorithm using single tournament selection with tournament size 7 to evolve permutations of 16 items to have a maximal number of reversals. The best possible fitness in this case is 120.*

Perform 100 runs each of the evolutionary algorithm for probabilities of 0%, 50%, and 100% of using the crossover operator and for using 0-1, 0-2, or 0-3 transposition mutations on each new permutation, selecting the number of mutations uniformly at random. Cut a

given run off after 10,000 mating events. Report the mean and standard deviation of time-to-solution, measured in mating events. Report the number of runs that timed out (did not find an optimal solution in 10,000 mating events). State any differences in the behavior of time-to-solution for the 9 different ways of using variation operators.

Maximizing reversals is a unimodal problem (see Problems 7.3 and 7.2). In fact, it is a very nice sort of unimodal problem in which not only is there only one hill, but there is a nonzero slope at every point. More precisely, any transposition must change the fitness of a permutation. The next experiment explores how much harder the problem of maximizing the number of reversals gets as n grows.

Experiment 7.2 *Run the software from Experiment 7.1 for all even permutation lengths from* $n = 8$ *to* $n = 20$ *using the best of the 9 settings for variation operators. Report the mean time-to-solution, the deviation of time-to-solution, and the number of failures. Perform the experiment again for population size 14 instead of 120. Discuss your results.*

Experiment 7.2 shows that the difficulty of maximizing reversals grows in a convex fashion, but not an unmanageable one. This is related to the very nice shape of the fitness landscape. Maximizing the order of a permutation has a less elegant fitness landscape. Let's repeat Experiment 7.1 for the problem of maximizing the order of a permutation.

Experiment 7.3 *Take the software from Experiment 7.1 and modify it to evolve permutations of maximal order. Evolve permutations of 20 items (which have a maximal order of 420 according to table 7.1). Leave the other parameters of the algorithm the same. Report the mean time-to-solution and standard deviation of the time-to-solution for each of the 9 ways of using variation operators. Compare with the results of Experiment 7.1, if they are available. Also, check the permutations that* achieve *the maximum and report the number of distinct solutions.*

Experiment 7.3 demonstrates that the problem of maximizing the order of a permutation interacts with the variation operators in a very different fashion from the problem of maximizing the number of reversals. Let's check and see if the difficulty of the problem increases smoothly with the number of items permuted.

Experiment 7.4 *Run the software from Experiment 7.3 for all permutation lengths from* $n = 20$ *to* $n = 30$*, using the best of the 9 settings for variation operators. Report the mean time-to-solution, the standard deviation of time-to-solution, and the number of failures. Get the correct maxima from Table 7.1. Does the problem difficulty grow directly as the number of items permuted?*

The preceding experiment should convince you that the fitness landscape for the problem of maximizing a permutation's order changes in an irregular manner as the number of items permuted grow. In the next experiment, we will try to learn more about the structure of the fitness landscape.

Experiment 7.5 *Review Definition 2.17 of stochastic hill climbers. Build a stochastic hill climber for locating permutations of high order with single transposition mutations that accepts new configurations that are no worse. Run the algorithm 100 times for 100,000 mutations per trial on permutations of length 22, 24, and 27. Report the fraction of the time that the final configuration was of maximal order and make a histogram of the frequency with which each order located was found. Explain the results.*

Experiment 7.5 uses software which can only climb up (or around) a hill. It has no ability to go downhill or to jump to another nearby hill. This means it can be used to locate local optima of the search space and give you some notion of how hard it is to fall into them. Unlike maximizing the number of reversals, the order maximization has places you can get stuck. In spite of this, Experiment 7.4 should have convinced you it is not a very hard problem. We will explore this juxtaposition of qualities in the Problems.

Random Key Encoding

It has already been noted that performing crossover on two permutations as if they were strings often yields non-permutations. One point partial preservation crossover is a way of crossing over permutations that works but seems, at least intuitively, to be a fairly disruptive operator. The issue of how to code a permutation in order to evolve it is another example of the representation issue. James Bean [8] came up with a pretty clever way of coding permutations as an array of real numbers like those used in Chapter 3. His technique is called a *random key* encoding.

In this case the word *key* is used to mean sorting key. A sorting key is a field in a database used to sort the records of the database. You might sort a customer database by customer's last name for billing and later by total sales to figure out who to have your salesmen visit. In the first instance, the name is the sorting key; in the second, total sales are the sorting key.

If we generate a list of n random real numbers then there is some permutation of those n numbers that will sort the numbers into ascending order. The random list of numbers, treated as a list of sorting keys, specifies a permutation. So: why bother? If we specify our permutations as the sorting order of lists of real numbers, then we can use the standard variation operators for arrays of real numbers that we defined in Chapter 3. This, at least, yields a new sort of crossover operator for permutations, and in some cases may yield superior performance. Let's do some examples to explain this new encoding for permutations.

Example 7.8 *For a selection of numbers placed somewhat randomly in an array, let's compute the permutation that sorts them into ascending order:*

Random key:	*0.7*	*1.2*	*0.6*	*4.5*	*-0.6*	*2.3*	*3.2*	*1.1*
Permutation:	*2*	*4*	*1*	*7*	*0*	*5*	*6*	*3*

Notice that this permutation simply labels each item with its rank.

Definition 7.12 *The* **random key encoding** *for permutations operates by storing a permutation implicitly as the sorting order of an array of numbers. The array of numbers is the structure on which crossover and mutation operate. The random key encoding is another representation for permutations.*

This encoding for permutations has some subtleties. First of all, notice that if two numbers in the random key are equal, then we have an ambiguous result. We will break ties by letting the first of the two equal numbers have the smaller rank. If we use a random number generator that generates random numbers in the range $(0, 1)$, then the probability of two numbers being equal is very close to zero, and so this isn't much of a problem.

A more important subtle feature of this encoding is its ability to have multiple inequivalent encodings for the same permutation. Notice we can change the value of a key by a small amount and not change the permutation. We simply avoid changing the relative order of the numbers. Such a small change could change a number enough to change where it showed up in a new permutation after crossover. In other words, when we use random key encoding of permutations, there are mutations that do not change the permutation, but which do change the results when the permutation undergoes crossover. This is not good or bad, just different. For a given problem, random key encoding may be better or worse than the direct encoding of permutations defined earlier in the chapter. Let's perform some comparisons.

Experiment 7.6 *Either write or obtain code that implements random key encoding for permutations. Use it instead of direct encoding with the software of Experiment 7.1. Initialize the random keys from the range* $(0, 1)$*. Use two point crossover of the random keys and uniform single point mutation of size 0.1. Perform the experiment with the random key modifications and compare with the results obtained using direct encoding. Discuss the pattern of differences in your write up.*

Since changing the representation to random key encoding should impact different problems to different degrees, we should test it on multiple problems.

Experiment 7.7 *Take the software from Experiment 7.6 and modify it to evolve permutations of maximal order. Evolve permutations of 20 items (which have a maximal order of 420 according to Table 7.1), but leave the other parameters of the algorithm the same. Report*

the mean time-to-solution and the standard deviation of the time-to-solution for each of the 9 settings for variation operators. Compare with the results of Experiment 7.3, if they are available. Also, check the permutations that achieve *the maximum and report the number of distinct solutions. Did the shift in the encoding make a noticeable change?*

We will look again at random key encodings in the later sections of this chapter and compare their performance on applied problems. There are a large number of possible encodings for permutations and we will examine another in the problems.

Problems

Problem 7.1 *What is the order of a permutation of n items that has a maximal number of reversals? Prove that the permutation in question, the one maximizing the number of reversals, is unique among permutations of* $0, 1, \ldots, n-1$.

Problem 7.2 *Perform the following study to help understand the fitness landscape for the problem of maximizing the number of reversals in a permutation. Generate 10,000 random permutations of 12 items and perform the following steps for each permutation: compute the number of reversals, perform a single transposition mutation, and compute the change in the number of reversals. Make a histogram of the number of reversals times the number of changes.*

Problem 7.3 *Prove the following statements:*

(i) A transposition must change the number of reversals by an odd number.

(ii) The maximum change in the number of reversals caused by a transposition applied to a permutation of n items is $2n-3$.

(iii) The fitness landscape for Experiment 7.1 is unimodal. Hint: *find an uphill path of transpositions from any permutation to the unique answer located in Problem 7.1.*

Problem 7.4 *Given the cycle type as an unordered partition $k_1, k_2, \ldots, k_m$ of n into positive pieces, find a formula for the number of permutations. (This is a very hard problem.)*

Problem 7.5 *List all possible cycle types of permutations of* $0, 1, \ldots, 5$ *together with the number of permutations with each cycle type. Recall that the total number of permutations is 5!=120.*

Problem 7.6 *Compute the number of cycle types for permutations of 10 items that yield permutations of order 6.*

Problem 7.7 *For n=3, 5, 7, 9, 11, 13, and 15, find a cycle type for a permutation that hits the maximum order given in Table 7.1.*

Problem 7.8 *Find the smallest n for which the cycle type of a permutation that hits the maximum order given in Table 7.1 is not unique - i.e. the smallest n for which there are multiple cycle types that attain the maximum order.*

Problem 7.9 *Is there a permutation that has the following properties:*

(i) The permutation does not have the largest possible order.

(ii) Every application of a transposition to the permutation yields a permutation with a smaller order.

Discuss such a permutation in the context of Experiment 7.5 whether or not it exists.

Problem 7.10 Essay. *Experiment 7.5 can be used to document the existence of local optima in the search space, when we are trying to find permutations of maximum order. In spite of the existence of these local optima, the evolutionary algorithm typically succeeds in finding a member of the global optima. Describe, to the best of your ability, the global optima of the search space. Hint: figure out why the maximum order is 420 for permutations of 19, 20, 21, and 22 items keeping firmly in mind that there is not a unique optimal solution.*

Problem 7.11 *Give an example of three random key encodings of permutations of 5 items such that the first two encode the same permutation, but, when crossed over with the second using the same two point crossover, they yield different permutations.*

Problem 7.12 *Is it possible to cross over two random key encodings of the same permutation and get new random key encodings of a different permutation? If your answer is yes, give an example; if your answer is no, offer a mathematical proof.*

Problem 7.13 *Consider the type of mutation used on random key encodings in Experiment 7.6. It is possible for such a mutation to fail to change the permutation that the random key encodes. True or false: if it does change the permutation, that change will be a transposition. Offer a proof of your answer.*

Problem 7.14 *Suppose that we have two mutation operators for random key encodings that behave as follows. The first picks a location at random and then finds the next smallest number to the one in that location, if it exists. If the number picked is the smallest, the mutation does nothing, otherwise it decreases the number in the chosen location by half the difference between it and the next smallest number. The second mutation is very like the first, save that it finds the next largest number and increases the number in the chosen location by half the difference between it and the next largest number. Notice that these mutations are neutral in the sense that they do not change the permutation encoded. What effect do they have?*

Problem 7.15 Essay. *As given in Experiment 7.6 the mutation operator permits the value of random keys to grow or shrink outside their initialized range. What effect does having a very large (or large negative) value have on the permutation the random key encodes? Discuss this in the context of the fitness functions of Experiments 7.6 and 7.7.*

Problem 7.16 Essay. *Suppose we do some large number of evolutionary algorithm runs for a problem using a random key encoding of permutations. Suppose that some positions within the permutation are uniformly large (or large negative values) across all populations, while others have small mean values over all populations. Discuss what, if any, useful information can be gleaned from these observations about solutions to the problem in general.*

Problem 7.17 *Prove that the initialization for permutations used in Experiment 7.6 is fair in the sense that all permutations have an equal chance of being selected.*

Problem 7.18 *Prove that any permutation of the set* $\{0, 1, \ldots, n-1\}$ *is the product of transpositions of the form* $(0\ i)$ *for* $1 \leq i \leq n-1$*. Let those transpositions be represented by the symbols* $a_1, a_2, \ldots, a_{n-1}$*. If we use strings over this collection of symbols, can we then evolve permutations using a string evolver?*

Problem 7.19 *For the representation of permutations given in Problem 7.18, compute the length of string needed to permit any permutation to be represented.*

Problem 7.20 *Is the encoding defined in Problem 7.18 fair, in the sense the word is used in Problem 7.17? Prove your answer.*

Problem 7.21 *The set of transpositions used to create an* $(n-1)$*-symbol representation for permutations in Problem 7.18 is an instance of a more general method. True or false: there is a binary string representation for permutations that uses only two permutations. This would require that all permutations be generated as a product of those two permutations repeated in some order. Prove your answer.*

7.2 The Traveling Salesman Problem

The Traveling Salesman problem is a classic optimization problem. It is typically phrased as follows: "Given a collection of cities for which the cost of travel between each pair of cities is known, find an ordering of the cities that lets a salesman visit the cities cyclically with minimum cost." If the cost of travel is dynamic, as with airline tickets, this problem can become difficult to even think about. The Traveling Salesman problem is usually abstracted into the following simpler form.

Definition 7.13 The Traveling Salesman Problem. *Given a set of points in the plane, find an order that minimizes the total distance involved in a circular tour of the points.*

Two examples of such tours, with integer-valued coordinates for the cities, are shown in Figure 7.2. Notice, in the second example that some very close links between pairs of cities are not used. The Traveling Salesman problem is an NP-hard problem and one of substantial economic importance. Such problems are typical applications of evolutionary computation. In this section, we will work with small examples with 16 or fewer cities in which the evolutionary algorithm is only slightly better than a brute force exhaustive search. In real world applications, the number of cities can range up into the thousands.

City	Position	City	Position
A	(1,93)	G	(12,39)
B	(45,75)	H	(47,38)
C	(29,18)	I	(8,27)
D	(87,18)	J	(88,73)
E	(50,5)	K	(50,75)
F	(23,98)	L	(98,75)

City	Position	City	Position
A	(10,90)	H	(32,33)
B	(48,35)	I	(28,60)
C	(76,50)	J	(98,85)
D	(56,35)	K	(10,10)
E	(34,20)	L	(34,52)
F	(68,52)	M	(1,92)
G	(42,28)		

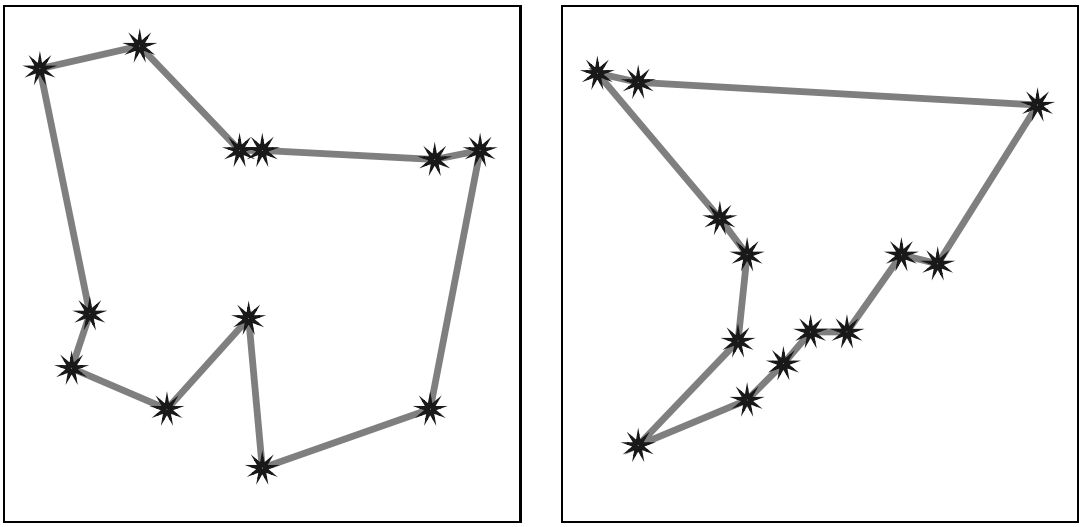


Figure 7.2: Coordinates for 2 Traveling Salesman tours and pictures of their corresponding optimal tours (The first has 12 cities that have an optimal Traveling Salesman tour with length 355.557; the second is comprised of 13 cities with optimal tour length 325.091.)

Different instances of the Traveling Salesman problem on the same number of cities can have varying difficulty. Compare the 13-city example in Figure 7.2 with one having the 13

cities arranged in a circle. The spike-like feature in the lower left of the picture of the tour contains cities that are both close together and which are not neighbors in the optimal tour. This means a simple algorithm that starts at a random city and then jumps to the closest city not already part of the tour could get stuck on this example, while it would solve 13 cities in a circle correctly no matter where it started. Let's check and see if these two cases look different to an evolutionary algorithm.

Before we build the evolutionary algorithm, there is one important point to make. For evaluating permutations as Traveling Salesman tours, we will use cycle notation, always with only one cycle. This looks very much like one-line notation, but isn't. Using one-line notation would not be a good idea, because it would be so easy to accidently mutate a tour into one with multiple independent cycles. Keeping this in mind, let's do out first Traveling Salesman experiment.

Experiment 7.8 *Take the software from Experiment 7.1 and modify the fitness function to be the length of the Traveling Salesman tour a permutation produces on the 13-city example given in Figure 7.2. For each of the 9 ways of using variation operators, report a 95% confidence interval on number of mutating events to solution. Selecting the best of these 9, rerun the software on a 13-city tour with the 13 cities placed equally around a circle of radius 45 centered at (50,50). You must compute the length of this tour yourself. Using 95% confidence intervals, compare the time-to-solution for the two problem cases.*

When possible, it is good to make immediate checks to see if representation has an impact on a given problem. Random key encoding is the "competing" encoding so far in this chapter. Let's check it on the current problem.

Experiment 7.9 *Modify the software from Experiment 7.8 to optionally use random key encoding. Using the combination of crossover probability and mutation type you found to work best, redo the experiment. Compare the 95% confidence intervals for the original and random key encodings. Is there a significant performance difference?*

In Experiments 7.2 and 7.4 we tried to estimate the degree to which a problem gets more difficult when made longer. For the Traveling Salesman problem, this is quite tricky, because the individual problems vary considerably in difficulty. Let's examine the degree to which the problem gets harder as it gets longer for a specific set of relatively easy instances, cities in a circle.

Experiment 7.10 *Use the software from Experiment 7.8 at its most efficient settings. For n =6, 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16, arrange n cities so that they are equally spaced around a circle of radius 45 with center (50,50). Note that* computing the length of the tour *is part of the work of setting up the experiment. Perform at least 100 runs per instance, more, if you can. Compute the mean time-to-solution with a 95% confidence interval. Plot the confidence intervals on the same set of axes. Check your results against Figure 7.3.*

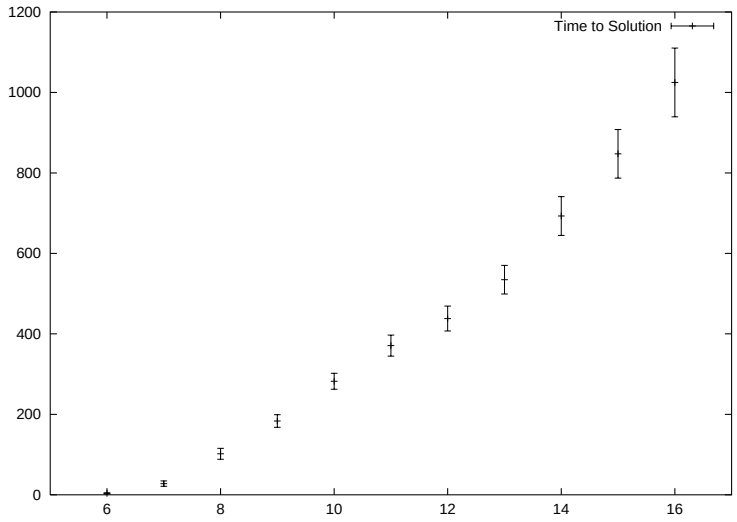


Figure 7.3: Confidence intervals for time-to-solution for simple circular tours used in Experiment 7.10

The difficulty of solving a given instance of the Traveling Salesman problem varies. Examine the solved examples shown in Table 7.2, which are visualized in Figure 7.4. Discuss, in class, if possible, what sorts of traps might be lurking in the examples that would make them especially difficult.

Experiment 7.11 *Before performing the computational portion of this experiment, predict which of the examples given in Table 7.2 are more or less difficult and your reason for thinking so. Using the software from Experiment 7.9 on its most efficient settings, perform 400 runs for each of the examples on 16 cities given in Table 7.2. Plot 95% confidence intervals for the time-to-solution. Discuss the result in light of your predictions.*

The issue of the level of difficulty of particular instances of the Traveling Salesman problem is a tricky one. What makes an instance difficult? The next experiment will take a stab at understanding this problem.

Experiment 7.12 *Use the software from Experiment 7.11 to perform 400 runs on 10 randomly generated problems with 12 cities. Generate city coordinates by randomly placing the cities with coordinates chosen uniformly at random in the interval [5,95]. Modify the software to use a population size of 1000 and extend the time before giving up to 20,000 mating events. With each random instance of the Traveling Salesman problem, run the software twice with a fixed random number seed. For the first run, set the length cutoff absurdly low. Use the minimum value found as the minimum tour length for the second set of runs. For*

Ex	Length	City coordinates			
A)	< 307	(57,20)	(93,6)	(33,76)	(86,83)
		(85,70)	(73,31)	(40,94)	(71,16)
		(66,67)	(14,46)	(65,40)	(78,32)
		(26,42)	(81,32)	(33,75)	(61,38)
B)	< 343	(85,20)	(54,21)	(5,70)	(40,32)
		(69,74)	(72,89)	(85,12)	(83,37)
		(86,44)	(78,30)	(42,51)	(80,35)
		(50,49)	(48,92)	(5,16)	(28,41)
C)	< 315	(66,79)	(89,14)	(88,25)	(51,63)
		(70,55)	(87,56)	(62,23)	(90,22)
		(31,61)	(49,61)	(71,22)	(11,90)
		(59,32)	(81,13)	(12,47)	(7,30)
D)	< 304	(91,44)	(91,44)	(15,10)	(58,43)
		(39,70)	(81,83)	(34,66)	(29,76)
		(84,35)	(40,52)	(73,22)	(92,60)
		(74,32)	(35,48)	(15,65)	(35,46)
E)	< 261	(95,50)	(83,64)	(64,64)	(56,65)
		(50,80)	(33,91)	(21,79)	(28,59)
		(35,50)	(28,41)	(21,21)	(33,9)
		(50,20)	(56,35)	(64,36)	(83,36)
F)	< 325	(95,50)	(72,59)	(64,64)	(67,91)
		(50,80)	(44,65)	(21,79)	(17,64)
		(35,50)	(17,36)	(21,21)	(44,35)
		(50,20)	(67,9)	(64,36)	(72,41)

Table 7.2: A collection of 16-city examples of the Traveling Salesman problem (Solutions are visualized in Figure 7.4.)

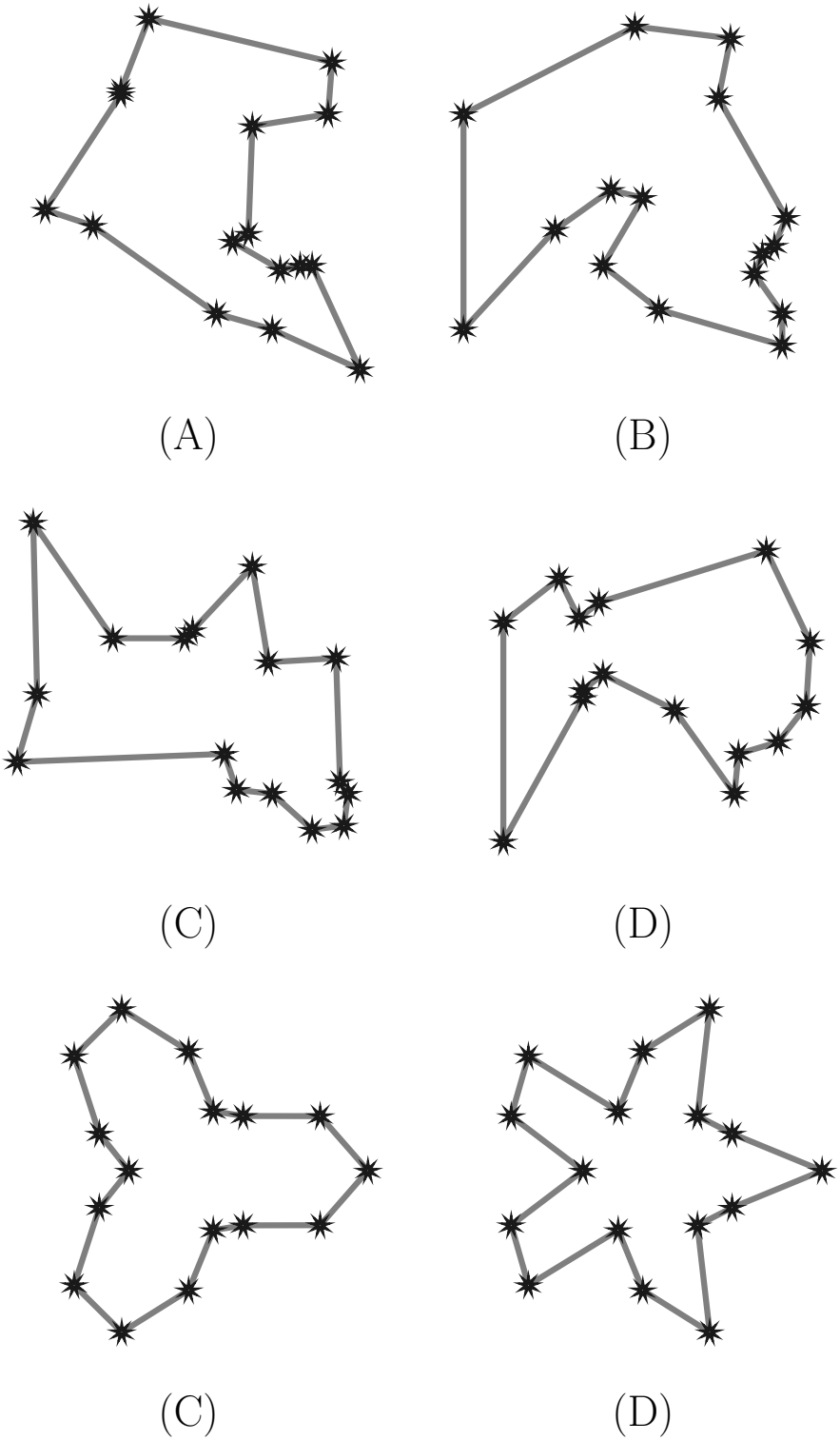


Figure 7.4: Visualizations of tour solutions for the test problems given in Table 7.2

each instance, plot 95% confidence intervals for the time-to-solution. Are there disjoint intervals? How did the random problem compare to the circle of 12 cities from Experiment 7.11? If this experiment is done by a class, pool and plot the examples.

Population seeding is a standard practice in evolutionary computation to generate smart initialization strategies. Population seeding consists of finding superior genes and incorporating them into the initial population. The superior genes are supposed to supply good pieces that can be mixed and matched by the algorithm. There is a real danger that population seeding will start the algorithm at a local optimum and get it stuck there. The superior genes may come from any source, and, in this case, we will use greedy algorithms to generate them.

Definition 7.14 *A* **greedy algorithm** *is one that makes an immediate choice that is the best possible without looking at the overall situation.*

Algorithm 7.1 Closest city heuristic

Input: A list of points in the plane
Output: A Traveling Salesman tour on the points
Details:
    Select a city at random to start the tour.
    Let the current end of the tour be the starting city.
    While cities remain:
        Find the city outside the tour closest to its end.
        Add that city to the tour as its new end.
    End(While)
    Complete the tour by connecting the first and last city.

Experiment 7.13 *Modify the software from Experiment 7.8 to use population seeding. Generate the closest city heuristic tour (Algorithm 7.1) for each of the 13 possible starting points and add those 13 tours into the starting population. Perform all 9 sets of runs for the different settings of variation operators. Did the population seeding help? Did a different combination of settings for variation operators work best?*

Random key encoding translates an array of random real numbers into a permutation. If we are going to use population seeding with random key encoding, we must be able to generate random key encodings that yield specific permutations. It is important that, while

doing this, we do not bias the random key. Recall that changes to the random key that do not change the permutation encoded can still have an effect (see Problems 7.11, 7.12). Before doing the next experiment, do Problem 7.29.

Experiment 7.14 *Modify the software from Experiment 7.13 to use random key encoding. Perform all 9 sets of runs for the different settings of variation operators. Compare your results with those from Experiments 7.8 and 7.13.*

A technique that is part of the standard toolkit for general problem solving is to break a problem into parts and solve it one piece at a time. Algorithm 7.1 is an example of this technique - you make a sequence of simple decisions. With the Traveling Salesman problem, the attempt to break up the problem into pieces makes some globally bad decisions, a topic we will explore in the Problems. Let's look at another method of breaking up the problem.

Algorithm 7.2 City insertion heuristic

Input: A list of points in the plane
Output: A Traveling Salesman tour on the points
Details:
    Select 3 cities at random and make a triangular tour.
    While cities remain outside the tour:
        Pick a city at random.
        For each adjacent pair in the tour,
            compute the sum of distances from the pair to the selected city.
        Insert the city between the adjacent pair with least joint distance.
    End(While)

Before we do an evolutionary experiment with the second heuristic, let's compare the performance of the two heuristics.

Experiment 7.15 *For the 6 tours on 16 cities given in Table 7.2, compare the performance of Algorithms 7.1 and 7.2. Run Algorithm 7.1 once for each possible starting city, and run Algorithm 7.2 1000 times on each example. Report the mean and best results for each algorithm.*

Now, with a sense of their relative merit as stand-alone heuristics, let us compare the two heuristics as population seeders.

Experiment 7.16 *Modify the software from Experiments 7.13 and 7.14 to do their population seeding using Algorithm 7.2. Use the crossover and mutation settings that worked best (which may be different for the two experiments), and perform the experiments again. Seed 13, 30, and 60 tours produced with Algorithm 7.2 into the initial population. The 13-seed runs permit direct comparison. Does use of the new seeding heuristic have an impact? Does the number of seeded population members have an impact? Explain your results.*

The Traveling Salesman problem is an applied problem and a hard problem. As Experiment 7.12 shows, the Traveling Salesman problem has a fitness landscape that changes from problem instance to problem instance. There is an enormous body of research on the Traveling Salesman problem, and many publications on evolutionary algorithms have the Traveling Salesman problem as their main focus. If the brief introduction in this section interests you, there is endless reading ahead. Making a real contribution to the Traveling Salesman literature is difficult (because so many clever people are there ahead of you), but worthwhile, because improvement in the algorithm has practical applications that save people money.

Problems

Problem 7.22 *Demonstrate that each optimal Traveling Salesman tour on n cities has $2n$ different versions in one-line notation. Give a normalization (a transformation of any of these tours into a unique representative) that picks out a unique way of writing the tour.*

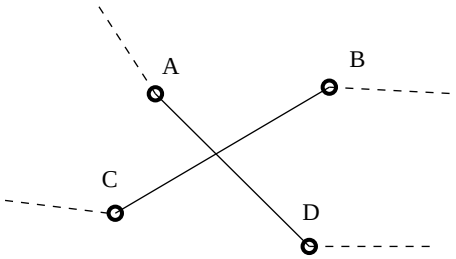
Problem 7.23 Essay. *As we say in the experiments in this section, the case of cities arranged in a circle is a very easy one. Real cities often are located in river valleys. Suppose we have a branching system of rivers with cities only along the rivers. Would you expect this sort of problem to be about as easy as the circle example, or much harder? Would you expect this sort of problem to be easier than, on a par with, or much harder than random problems generated in a manner similar to that used in Experiment 7.12.*

Problem 7.24 *Suppose that we start with 4 cities in the corners of a square with side length= 1 and coordinates (0,0), (0,1), (1,0), (1,1). Consider a 5th city anywhere in the Cartesian plane with coordinates (x, y). Let $f(x, y)$ be the length of an optimal Traveling Salesman tour on the 5 cities. Answer the following questions.*

- *Is $f(x, y)$ continuous?*
- *Describe for which points $f(x, y)$ is differentiable and for which it is not.*
- *Sketch the set of points where $f(x, y)$ is not differentiable.*

Problem 7.25 *Build an exhaustive searcher that can locate the true minimal tour length for a given collection of cities. Hint: use the results of Problem 7.22. Generate 1000 random examples with 8 cities that have their coordinates chosen uniformly at random from the interval [5,95], and compute their correct minimal tour length by exhaustion. Plot the results as a histogram. What type of distribution is it?*

Problem 7.26 *Suppose we were to place cities uniformly at random within a circle of radius R. Give and defend a bound on the maximum length of the shortest tour involving both the radius and the number of cities. This is a funny problem in that we want to know the maximum, over all possible examples, of the minimum tour length. You may want to try some numerical examples, but your final answer must be defended by logic in a mathematical fashion. (This is a hard problem.)*



Problem 7.27 *Suppose that 4 cities are configured as shown above as part of a Traveling Salesman tour. Prove that the other two ways to connect the 4 cities (A-B C-D and A-C B-D) yield shorter total distance for the two trip legs involving these 4 cities. Is it always possible to change to one of the other two connections of A, B, C, and D* without *breaking the tour into two pieces?*

Problem 7.28 *Suppose you have a predicate that can detect the crossing of two line segments. Describe a mutation operator based on Problem 7.27, and perform a cost/benefit analysis in a few sentences.*

Problem 7.29 *Verify that the following procedure for converting a permutation into a random key is correct and unbiased. Assume we are creating a random key to encode a permutation $f(i)$ on n items. Generate an array of n uniformly distributed random numbers and sort it. The random key will use the values in this array. Going through the permutation, place the ith item of the sorted array into the $f(i)$th position in the random key. You must show two things: that this procedure generates a random key that encodes $f(i)$ (is correct), and that the key is chosen uniformly at random from among those that encode $f(i)$ (is unbiased). If either of these properties does not hold, document the flaw and repair the procedure.*

Problem 7.30 *Consider the following collection of points in the plane:*

$$(0,0),(2,2),(4,0),(6,2),(8,0),(10,2),(12,0),(14,2).$$

Compute the tour resulting from applying Algorithm 7.1, and, also, compute the optimal tour.

Problem 7.31 *Implement Algorithm 7.2 and run it several times on the collection of points given in Problem 7.30. Compare with the optimal tour and the result of using Algorithm 7.1.*

Problem 7.32 *What is the largest number of distinct answers that Algorithm 7.1 produces for a given n-city tour selected at random?*

Problem 7.33 Essay. *Contrast and compare Algorithms 7.1 and 7.2. Address the following issues. Are there obvious mistakes that each of these algorithms makes? Which of these algorithms are deterministic? Which of these algorithms exhibits superior performance* on its own*? Does superior stand-alone performance lead directly to improved performance in a seeded evolutionary algorithm?*

Problem 7.34 Essay. *There is a saying that "you can't make money without spending money," and, in fact, most methods of making a lot of money require investment capital of some sort. A counterexample to this principle is the fortune of J. K. Rowlings, author of the fantastically popular* Harry Potter *series. At the time of this writing, the author's fortune exceeds that of the Queen of England, and yet the total investment in producing the first book was quite modest. Keeping these two different notions of wealth creation in mind, comment on the worth and impact of improving performance on the Traveling Salesman problem. There is economic impact from such improvements: what sort of capital investment do such improvements require? Is the process of improving algorithmic performance on the Traveling Salesman problem more like that of a venture capitalist or a successful author? Explain.*

Problem 7.35 *Create and test your own Traveling Salesman heuristic as a stand-alone tool using the solved examples in this section. Compare it with the two heuristics given.*

7.3 Packing Things

This section will treat the problem of packing generic objects in generic containers. The packing will be performed by a greedy algorithm that is controlled by an evolved permutation. This problem is easier, on average, than the Traveling Salesman problem because an optimal solution often has many forms. Let's describe the basic greedy algorithm that we will subsequently modify into a fitness function.

Algorithm 7.3 Greedy Packing Algorithm

Input: A set of n indivisible objects G of size $g_1, g_2, \ldots, g_n$ and a set C of containers of size $c_1, c_2, \ldots, c_k$
Output: An assignment of objects to containers and a list of objects that fit in no container
Details:

Taking the objects in the order presented, assign an item to the first container with sufficient space remaining for it. When an object is assigned to a container, subtract the item's size from the container's capacity. Goods that cannot fit in a container, when their turn comes up to be assigned, are put in the group that fit in no container.

Definition 7.15 *The* **excess** *is the sum of the sizes of the objects that fit in no container.*

In order to make the algorithm clear, we will trace it in an example.

Example 7.9 *Suppose we have 6 objects of size 20, 40, 10, 50, 40, and 40 and 2 containers of size 100 and 100. Then, as we assign the objects to the containers, the remaining capacity in the containers will behave in the following fashion:*

Good number	*Capacity 1*	*Capacity 2*	*Excess*	*Assigned to*
-start-	*100*	*100*	*0*	
1	*80*	*100*	*0*	*1*
2	*40*	*100*	*0*	*1*
3	*30*	*100*	*0*	*1*
4	*30*	*50*	*0*	*2*
5	*30*	*10*	*0*	*2*
6	*30*	*10*	*40*	*excess*

Notice that the greedy algorithm did not do a good job. Since 20+40+40=100 and 10+40+50=100, all the objects could have been packed (and would have if they had been presented in a different order).

With this example in hand, it is possible to describe a fitness function for permutations that permits us to search for good assignments of objects to containers. In the example, we could have achieved a better packing by presenting the objects in a different order. Since permutations specify orders, we can improve the greedy algorithm by having a permutation reorder the objects before they are presented to it.

Definition 7.16 *The* **greedy packing fitness** *of a permutation σ for a given set of objects $g_1, g_2, \ldots, g_n$ and a given set of containers of size $c_1, c_2, \ldots, c_n$ is the excess that results when the algorithm is applied with the objects presented in the order $g_{\sigma(1)}, g_{\sigma(2)}, \ldots, g_{\sigma(n)}$. This fitness function is to be minimized for efficient packing.*

At this point, we need some example problems. A problem consists of a number of containers, the sizes of those containers, a number of objects, and the sizes of those objects. Table 7.3 gives 9 examples chosen for varying size and difficulty. In general, size and difficulty increase together, but not always. While there is no requirement that the objects available exactly fill the containers, all the examples given in Table 7.3 have this property. Let's start by getting a handle on the relative difficulty of these experimental cases.

Experiment 7.17 *For each of the 9 experimental cases given in Table 7.3, build or obtain software to randomly generate permutations until one is found with a greedy packing fitness of 0. One run of this software is called a* trial. *Write the software to stop, if it must examine more than 20,000 permutations. If no permutation is found that has an excess of 0 after 20,000 permutations are examined, record the trial as a failure; otherwise, record it as a success. The number of permutations examined, before a trial succeeds, is its length. The length of failures is undefined. For each of the 9 cases given, perform 400 trials. Report the number of failures and the mean and standard deviation of the lengths of successful trials.*

Experiment 7.18 *Modify the code from Experiment 7.17 to perform trials in a different manner: as a stochastic hill climber (see Chapter 2, Definition 2.17). Starting with a random permutation, the software should apply a mutation and save the result, if it is no worse (has no larger excess) than the current permutation. For each of the cases given in Table 7.3, run 4 different stochastic hill climbers, 400 times each. The 4 stochastic hill climbers should use (i) the standard representation with single transposition mutation, (ii) the standard representation with double transposition mutation, (iii) the random key encoding with single point mutation as in Experiment 7.6, and (iv) the random key encoding with two point mutation. Record the number of successes and failures for each case and type of stochastic hill climber.*

As we saw in Section 2.6, the stochastic hill climber can be used to estimate the roughness of the "terrain" in a fitness landscape. Since the current permutation is never allowed to increase its excess, the hill climber tends to find the top of whatever hill it started on. The key observation is that the hills *do not* exist in the space of permutations alone. Rather the hills are created by the connectivity induced by the mutation operators. This means Experiment 7.18 may well have predictive value for the best representation and mutation operator to use in our evolutionary algorithms.

Case	Containers	Sizes	Goods	Sizes
1	3	100(3)	12	39, 37, 34, 33, 28, 25, 23, 22, 19, 17, 15, 8
2	4	100(4)	14	60, 40, 39, 37, 34, 33, 28, 25, 23, 22, 19, 17, 15, 8
3	5	100(5)	23	60, 55, 50, 45, 40, 30, 25(2), 20, 13(2), 12(2), 11(5), 9(5)
4	6	100(6)	20	56, 55, 54, 53, 52, 51, 49, 24(3), 23(3), 22(2), 9(5)
5	7	100(7)	22	60, 59, 58, 57, 56, 49, 48, 46, 36, 27, 26, 25, 24, 22, 21, 20, 19, 15, 14, 7, 6, 5
6	7	100(7)	22	60, 59, 58, 57, 56, 49, 48, 46, 36, 27, 26, 25, 24, 22, 21, 20, 19, 15, 14, 7, 6, 5
7	7	100(7)	22	65, 56(2), 55, 54, 53, 52, 47, 38, 25, 24(2), 23(3), 22, 13, 11, 9(4)
8	6	100(5), 200	22	65, 56(2), 55, 54, 53, 52, 47, 38, 25, 24(2), 23(3), 22, 13, 11, 9(4)
9	5	100(3), 200(2)	22	65, 56(2), 55, 54, 53, 52, 47, 38, 25, 24(2), 23(3), 22, 13, 11, 9(4)

Table 7.3: Examples of containers and objects for the Packing Problem (Numbers in parentheses indicate the number of containers or objects of a given size.)

Experiment 7.19 *Modify the code from earlier experiments that evolve permutations using either the standard representation of permutations with one point partial preservation crossover and one point transposition mutation, or using the random key encoding with two point crossover and single point mutation as in Experiment 7.6. Using all 4 types of evolutionary algorithms on Cases 1-5 of the Packing Problem given in Table 7.3, compute the mean time-to-solution, standard deviation of time-to-solution, and the number of times the algorithms failed to find a solution. Use initial populations of 200 permutations and evolve for at most 100,000 mating events using a steady state algorithm. Use single tournament selection with tournament size 7. Compare the two methods. If you performed Experiment 7.18, check and see if the stochastic hill climber was predictive of the behavior of the evolutionary algorithms.*

Cases 1-5 in Table 7.3 can all be solved in a reasonable amount of time by the stochastic hill climber (if it hits the right hill), and so are reasonable targets for any of the evolutionary algorithms in Experiment 7.19. Cases 6-9 are all variations of one basic problem that explore variation of the distribution of objects and sizes of containers.

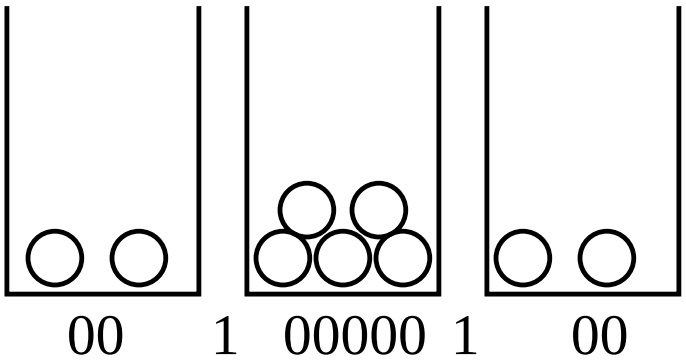
Experiment 7.20 *Pick the evolutionary algorithm from Experiment 7.19 that turned in the best performance. Use that code on Cases 6-9 of the Packing Problem given in Table 7.3. Record the same data as in Experiment 7.19. Discuss the issue: "Is there variable hardness in a random problem case?"*

Experiment 7.20 demonstrates that there is a difference in the difficulty of different cases of the Packing Problem. It would be interesting to see if we can locate difficult and simple instances of the problem automatically. To do this, we need the ability to rapidly generate instances of the Packing Problem. This is related to a well-known combinatorial problem of placing balls in bins. Suppose that we have n adjacent bins and want to place k indistinguishable balls in those bins. The problem is to enumerate the space of possible configurations. In a combinatorics course, we would simply count the configurations; here we want a representation that permits us to search (or even evolve) Packing Problems.

Theorem 7.5 *There are* $\binom{n+k-1}{k}$ *ways of placing* k *indistinguishable balls into* n *adjacent bins. Each configuration corresponds to a binary word with* $n-1$ *ones and* k *zeros.*

Proof: The zeros represent balls. Zeros before the first one represent balls in the first bin. Each one represents the division between two bins. Zeros between two ones are all in the bin "between" those ones. The zeros after the last one in a given binary string are the balls placed in the last bin. The number of such strings is equal to the sum of the number of ones and zeros choose the number of zeros. □

Example 7.10 *Suppose we have 9 balls and 3 bins. Then, the correspondence given in Theorem 7.5 maps the string* **00100000100** *to the ball-bin configuration:*



The 1s divide the containers from one another. 3 containers yield 2 boundaries between containers.

What does filling bins with indistinguishable balls have to do with creating examples of the Packing Problem? We will use the ball-bin problem to help us generate a list of sizes

for objects, given a list of sizes of containers. We can limit ourselves to the problem of finding a list of object sizes that exactly fill the containers. If we want examples that overfill the containers, we can add a few objects; if we want excess capacity, we can remove a few objects. Cases with exact solutions, thus, are the starting points from which we can generate any possible case.

Assuming we have a list $c_1, c_2, \ldots, c_k$ of container sizes, we treat finding the objects to go in each container as a single balls-in-bins problem. The number of balls is equal to the capacity of the bins k, while the number of objects n is equal to the number of bins. Generating a random binary string with k balls (zeros) and $n-1$ bins (ones) thus specifies objects to fill a single container. Repeating this procedure for each container, gives a set of object sizes that fill the container exactly. Generating a random binary string with a specified number of ones and zeros is similar to generating the standard representation of a permutation. Put the correct number of ones and zeros in an array, and then repeatedly swap randomly chosen pairs of items in the array.

Experiment 7.21 *Create or obtain code that implements the problem case generator described above. The generator should save, not only the sizes of the objects, but the binary strings used in creating the case. Use 5 containers of size 100 with 4 objects per container.*

For a given case, count the number of permutations out of 100,000 selected at random that correctly solve it. This is its hardness *(low numbers are hard). Generate 2000 problem cases and give the distribution of hardnesses as a histogram with 20 bars, each with a width of 5000 hardness units. If you performed experiment 7.19 or 7.20, run that software on the hardest and easiest problem case you generated in this experiment.*

For the Traveling Salesman problem, we examined the effect of seeding the population with superior solutions generated by heuristics. It would be interesting to see if that procedure helps solve harder cases of the Packing Problem. We will use a very simple heuristic.

Experiment 7.22 *Pick the evolutionary algorithm from Experiment 7.19 that turned in the best performance. Modify the part of the code that creates the initial population, so that, rather than using 200 permutations generated randomly, you use the best 200 out of 400, 800, and 1200. Record the mean time-to-solution, the standard deviation of time-to-solution, and the number of times the algorithms failed to find a solution for Cases 6-9 of the Packing Problem given in Table 7.3, as well as the hardest case you located in Experiment 7.21 (if you performed that experiment). Does the population seeding help? If so, does it help enough to justify its cost? Is the effect uniform over the problem cases? Do the experiments suggest that the filtration of even larger initial groups would be a good idea for any of the problem cases?*

A class activity that is interesting is to compete to construct difficult cases of the Packing Problem. Each student should turn in a case together with a solution (certificate of

solvability). These cases should then be compiled by the instructor and given to all students (with the names removed). Each student then attempts to solve each of the problem cases with evolution code (any version). Students are scored on the number of other students that fail to solve their case.

Problems

Problem 7.36 *How many different ways can you completely fill a container of size 100 with n objects? Clearly, there is only one way to do it with 1 object (the size of the object must be 100). If we assume the objects have integer sizes, then there is only one way to fill the container with 100 objects - use objects of size 1. How many ways are there, if we do not care about the order in which the objects are put into the container, to fill it with* $n = 2, 3,$ *or* 4 *integer-sized objects? Hint: don't do this by hand. (This is a type of mathematical problem called* partitioning*).*

Problem 7.37 *Enumerate all the orderings of the objects in Example 7.9 that permit all the objects to be packed by the greedy algorithm. An explicit listing may be sort of long; be clever in how you enumerate.*

Problem 7.38 *In many of the examples in this section we have only one size of container. For examples which have multiple container sizes, would it help to permute the order in which the containers are filled? Prove that your answer is correct.*

Problem 7.39 *A case of the Packing Problem is* tight*, if the sum of the sizes of the objects exactly equals the sum of the sizes of the containers. Give an example of a tight problem, with 3 containers of size 100, that is impossible - there is no way to fit all the objects in the bins.*

Problem 7.40 *Reread problem 7.39. Construct a tight problem with 3 containers of size 100 that has an excess of 0 for* every *permutation of the presentation order of its objects.*

Problem 7.41 *Assume that objects and containers have integer sizes. Prove that every Packing Problem in which the sizes of the containers and objects have a common divisor larger than one is equivalent to a problem in which they do not.*

Problem 7.42 *Draw the balls-in-bins diagrams for the following binary strings in the same style as Example 7.10.*

(i) **010**

(ii) **0010001000010**

(iii) **10001000100**

(iv) **001100**

(v) **0110100100110**

Problem 7.43 *On page 188, there is an informal description, but not pseudo-code for an algorithm for generating random problem cases for the Packing Problem. Carefully write out this pseudo-code.*

Problem 7.44 *Generalize the random case generator to also generate, using a trinary string, the container capacities. As 1s were boundaries between bins, 2s become boundaries between containers. Be sure to give an interpretation for all possibilities, such as adjacent 1s or 2s.*

Problem 7.45 *It would be interesting to know, for a given case of the Packing Problem, how many different solutions there are. A problem with zero solutions is, in some sense, maximally hard. A solvable problem is probably easier, if there are many solutions. Give and defend a definition of* different solutions. *Answer the following questions, as part of your defense. Is it enough for the permutations controlling the greedy packing algorithm to be different? Is it enough for a given object to be in a different container? Does being different require that the numbers added to fill a container not appear in the solution being judged to be different?*

Problem 7.46 Essay. *Suppose that we use a string evolver to attack the Packing Problem. The alphabet will contain one character for each container and an additional character for the placement of an object in no container. A given string will have a length equal to the number of objects. The ith character gives the container assignment of the ith object. Both the excess (total size of objects assigned to no container) and the number of containers overfilled (containers in which no more objects will fit) are computed. When strings are compared, having fewer containers overfilled is more important than having less excess. These two numbers are used in a lexical product fitness function. Your task: write an essay that estimates the relative performance of this evolutionary algorithm and one of the algorithms used in this section. (Implementing the algorithm would provide the best form of evidence for your essay, but is not required.)*

Problem 7.47 *The containers used in this section have a pretty boring sort of geometry. They simply have a one-dimensional capacity that is used up as objects are placed in them. Suppose that we have containers that are $N \times M$ rectangles and objects that are $H \times K$ rectangles. Rewrite the greedy packing fitness for placing rectangular objects into rectangular containers so that they fill the containers efficiently. Modify the permutation data structure,*

if you deem it necessary. Since the objects have two orientations in which they may be placed, you may need to augment your permutations with some sort of orientation switch. Write pseudo-code for the new fitness function.

Problem 7.48 *The hardness criterion given in Experiment 7.21 is sort of slow to use as a fitness function for evolving hard (or easy) problem cases. Give and defend a better fitness function for evolving hard problem cases.*

7.4 Costas Arrays

This section develops the theory needed to attack an unsolved mathematical problem, the existence of a Costas array of order 32.

Definition 7.17 *A* **Costas array** *is an $n \times n$ array with one dot in each row and column and the property that vectors connecting any two dots in the array do not occur more than once (i.e., all such vectors have different lengths or different slopes). The number n is the* **order** *of the array.*

Costas arrays were originally devised as signal processing masks John P. Costas. Given that these arrays are useful for signal processing, the next mathematical question is "do they exist?" For infinitely many n, the answer is known to be yes, but, for some n, the answer is not yet known. The $n < 100$ for which no Costas array is known to exist at the time of this writing are: **32, 33, 43, 48, 49, 54, 63, 73, 74, 83, 84, 85, 89, 90, 91, 92, 93,** and **97**. The total number of Costas arrays of order n is a very odd function of n. Examine Table 7.4 to gain some sense of this oddity. Examples of Costas arrays of order 10 and 12 are given in Figure 7.5. Before continuing on with Costas arrays, we need to review some useful linear algebra.

If we have an $n \times n$ matrix M, then the process of multiplying a vector $\vec{v}$ by M creates a map from $\mathbb{R}^n$ to itself. If M is a matrix with a single one in each row and column and all other entries zero, it acts on $\mathbb{R}^n$ by permuting the dimensions.

Definition 7.18 *A* **permutation matrix** *is a matrix with a single 1 in each row and column and all other entries 0.*

Assume $\vec{v}$ is a row vector and M is a permutation matrix. To multiply M by $\vec{v}$, we could compute either $M \cdot \vec{v}^t$ or $\vec{v} \cdot M$. The resulting permutations are inverses of each other. Let's do an example.

Example 7.11 *Suppose that*

$$M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Dimension	Number	Dimension	Number
1	1	2	2
3	4	4	12
5	40	6	116
7	200	8	444
9	760	10	2160
11	4368	12	7852
13	12828	14	17258
15	19612	16	21104
17	18276	18	15096
19	10240	20	6464
21	3536	22	2052
23	872	24	? (> 1)

Table 7.4: Costas arrays of those sizes for which the total number of Costas arrays is known

and that $\vec{v} = (a, b, c)$. *Then*

$$M \cdot \vec{v}^t = (b, c, a),$$

while

$$\vec{v} \cdot M = (c, a, b).$$

The permutations (b c a) and (c a b), in one-line notation, are inverses of one another.

The reason for introducing permutation matrices at this point is that the "one-per-row-and-column" condition is common between Costas arrays and permutation matrices. To evolve Costas arrays, we will take the machinery already developed for evolving permutations, add a map from permutations to permutation matrices, and then search the space of permutation matrices for those that obey the additional condition needed to be Costas arrays. (To do this, we will need to substitute dots for 1s and blanks for 0s.)

Definition 7.19 *For a permutation* σ *of* $\{1, 2, \ldots, n\}$ *let the* **permutation matrix associated with** σ, M_σ, *be the permutation matrix with ones in position* $(i, \sigma(i))$, $i = 1, 2, \ldots, n$.

Example 7.12 *If* $\sigma = (12453)$ *in cycle notation, then*

$$M_\sigma = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

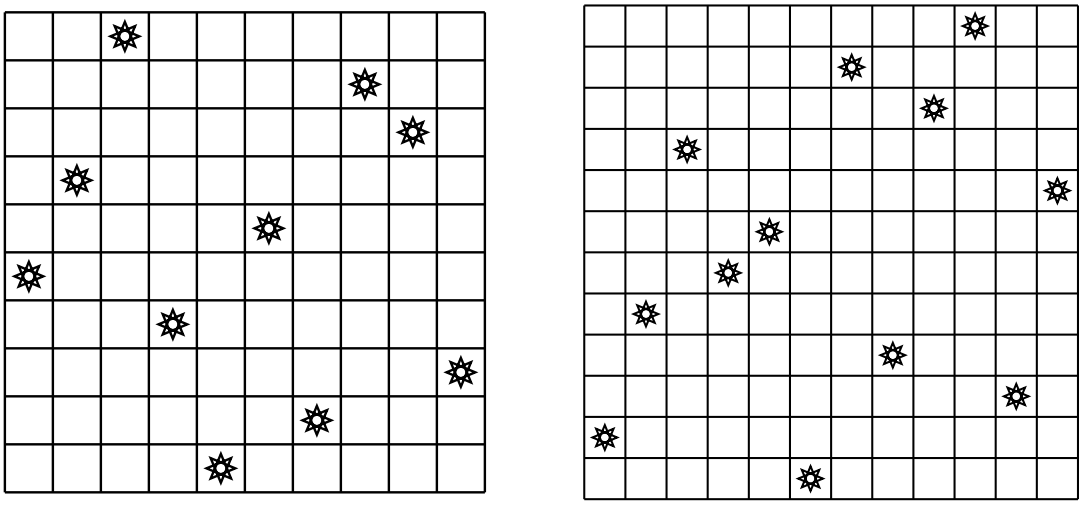
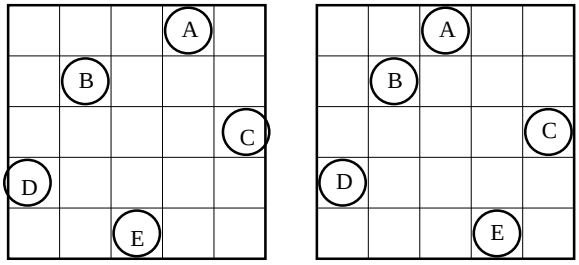


Figure 7.5: Matrix presentations of Costas arrays of order 10 and 12

The map from permutations to potential Costas arrays is useful for displaying the final array in the form of a Costas array. The fundamental structure undergoing evolution, however, is the permutation, and the fitness function we will use is based on the permutation in one-line notation. Let's examine two permutations, one of which is a Costas array, in order to understand the additional property of vectors being unique.

Example 7.13 *Shown below are a pair of* 5×5 *arrays with one dot in each row and column. The first is a Costas array, the 10 vectors joining the pairs of dots AB, AC, AD, AE, BC, BD, BE, CD, CE, and DE are all different in their* x *or* y *length or both. In the second array the vectors joining BC and DE are identical. Notice that we do not distinguish between* $\vec{BC}$ *and* $\vec{CB}$*.*



The permutation for the first array, in cycle notation, is (1 4)(2)(3 5) while the second array is associated with the permutation (1 3 5 4)(2). In one-line notation these permutations would be (4 2 5 1 3) and (3 2 5 1 4).

We need a fitness function that will favor Costas arrays over mere permutation matrices. Given that Costas arrays require vectors connecting dots in the matrix to be unique, there is a natural choice: count the number of reused vectors (and minimize).

Definition 7.20 *The* **number of violations of the Costas condition** *in a permutation matrix is the number of times that a vector is reused. A vector used 3 times is reused twice and so contributes 2 to the total. The* **VCC fitness function** *of a permutation σ is the number of violations of the Costas condition that appear in M_σ, the corresponding permutation matrix.*

The definition of the VCC fitness function does not immediately suggest a simple algorithm for computing it. If we list the permutations in one-line notation, then dots in adjacent columns of M_σ are adjacent in σ, dots two apart horizontally in M_σ are two apart in σ, and so on. So, we can count violations of the Costas condition by looking for repetitions in the distance i differences of the permutation.

Example 7.14 *Compute the number of violations of the Costas Condition in the permutation $\sigma =$***(3 4 5 0 1 2)**.

Permutation:	*3*	*4*	*5*	*0*	*1*	*2*	*Repetitions*
$\sigma(i+1)-\sigma(i)$	*1*	*1*	*-5*	*1*	*1*		*3*
$\sigma(i+2)-\sigma(i)$	*2*	*-4*	*-4*	*2*			*2*
$\sigma(i+3)-\sigma(i)$	*-3*	*-3*	*-3*				*2*
$\sigma(i+4)-\sigma(i)$	*-2*	*-2*					*1*
$\sigma(i+5)-\sigma(i)$	*-1*						*0*

Totalling the repetitions for each horizontal component length, we have 3+2+2+1+0=8 violations. Notice that, since the permutation does not wrap around, there are fewer vectors with long horizontal components than short ones. In a permutation of 6 numbers, there is only one vector with horizontal component 5 (in this case, its vertical component is -1), and so there is no chance of repetition there.

Experiment 7.23 *Using a random key encoding for permutations in one-line notation, build or obtain software for an evolutionary algorithm for locating Costas arrays. Initialize the random key to have its numbers in the range $[0, n]$ where n is the order of the Costas arrays being searched. Let your point mutation be uniform real mutation with size 0.5. Test the software with one, two, and three point mutation and two point crossover for Costas arrays of order 12. Use a population of 1000 permutations. Compute the mean time until an array is located and the standard deviation of this time over 400 trials. Have the algorithm stop after 200,000 fitness evaluations and declare an algorithm that times out a failure. Record the number of failures. Compare the 3 different mutation operators. Do the data indicate that trying four point mutation might be advisable?*

The number of mutations is one possible parameter that governs the mutation-based part of the evolutionary search. Another is the size of the mutations.

Experiment 7.24 *Perform Experiment 7.23 again, only this time fix the number of mutations at whatever worked best (or two if you didn't do Experiment 7.24). Rather than varying the number of mutations, vary their size. Use mutation sizes 0.25, 0.5, 1.0, and 2.0, pulling in the data from 0.5 from the previous experiment, if it is available.*

The crossover operator also has some impact on the behavior of a system that evolves permutations, and so may be worth examining. Since Costas arrays are connected with unsolved problems, we will cast a fairly wide net, going beyond the obvious possibility of playing with the number of crossover points to the extremity of introducing a new type of crossover operator.

Definition 7.21 Non-aligned crossover *for a pair of genes organized as a string or an array is performed in the following manner. A length smaller than or equal to the length of the shorter two participating strings or arrays is chosen. Starting positions in both strings are chosen. Going down the strings we then exchange characters in substrings or sub-arrays of the selected length, wrapping if we go off the end of the string.*

The standard crossover operator contains a tacit agreement that each position within the objects being crossed over has a particular meaning. When these meanings are distinct, using non-aligned crossover would mean comparing apples and oranges. With random key encoding, the relative rank of the numbers in the array is what matters, not their positions. This means that all meaning is between numbers, and, so, non-aligned crossover is, perhaps, more meaningful.

Experiment 7.25 *Perform Experiment 7.24 again, only this time fix the number and size of mutations at whatever worked best (two point and 1.0, for example). Instead of testing different mutation operators, test one point crossover and two point crossover. Use both the standard and non-aligned forms of the crossover with the probability of non-aligned crossover being 0%, 10%, and 25% in different runs. This requires 6 different sets of runs. Does non-aligned crossover help? Might upping the percentage be a good thing? Warning: be sure your random key encoding software deals consistently with ties.*

Experiments 7.23-7.25 all tuned up our search system on order 12 Costas arrays. It is now time to see how far the system can be pushed.

Experiment 7.26 *Perform Experiment 7.25 again, using the collection of settings that you found to work best. Allocating runtime on available machines so as to avoid actually getting into trouble or breaking rules, find the largest order Costas array you can with your software.*

Run experiments on several orders to obtain a baseline behavior before doing the big run. Keep in mind that the absolute number of Costas arrays declines at order 17, even though the search space (or number of permutations) is growing at a faster than exponential rate (in fact, factorially). This means that time estimates for orders below 17 will have no meaning for those over 17. If you find a Costas array of one of the unknown orders, publish the result and send the author of this book a copy of the manuscript, please.

On page 191, the claim was made that an infinite number of n are known for which there are Costas arrays. Such arrays may be useful in seeding initial populations (or they may not).

Definition 7.22 *For a prime number p, a number k $(mod\ p)$ is said to be* **primitive** *$(mod\ p)$, if every nonzero value $(mod\ p)$ is a power of k.*

Example 7.15 *Let's look at all powers of all nonzero numbers $(mod\ 7)$. Recall that the powers cycle after the 6th power (and sooner in elements that are not primitive).*

	Power					
k	*1*	*2*	*3*	*4*	*5*	*6*
1	*1*	*1*	*1*	*1*	*1*	*1*
2	*2*	*4*	*1*	*2*	*4*	*1*
3	*3*	*2*	*6*	*4*	*5*	*1*
4	*4*	*2*	*1*	*4*	*2*	*1*
5	*5*	*4*	*6*	*2*	*3*	*1*
6	*6*	*1*	*6*	*1*	*6*	*1*

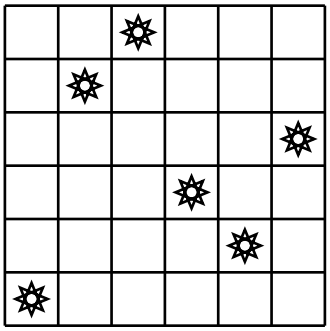
As the table shows, the only two primitive numbers $(mod\ 7)$ are 3 and 5. There are many, many patterns in the above table and most books on group theory or number theory both generalize and explain these patterns.

It turns out that primitive numbers can be used to generate Costas arrays.

Theorem 7.6 *Let k be a primitive number $(mod\ p)$ for some prime p. Then the permutation σ on $\{1, 2, \ldots, (p-1)\}$ for which $\sigma(i) = k^i (mod\ p)$ has the property that M_σ is a Costas array.*

Proof: is left as an exercise, Problem 7.54.

Example 7.16 *Notice that the construction given in Theorem 7.6 yields permutations of $p-1$ items that don't include zero. Lets look at the Costas array generated by 3 $(mod\ 7)$. Using Example 7.15, we see that σ makes the following assignments: $1 \to 3$, $2 \to 2$, $3 \to 6$, $4 \to 4$, $5 \to 5$, and $6 \to 1$. This corresponds to the permutation matrix:*



Let's conclude with an experiment in population seeding.

Experiment 7.27 *Perform Experiment 7.26 again at the maximum order for which you were able to find a Costas array or the next smallest order, whichever is not one less than a prime. The initial population should be generated in the normal fashion and also by taking permutations of the sort described in Theorem 7.6 and randomly extending them to the desired order. Invent and describe the means you use to extend these permutations. Be* very *sure your extended items are still permutations. Does seeding the population in this fashion help?*

Problems

Problem 7.49 *Prove that if we apply any of the 8 symmetries of the square to a Costas array, it remains a Costas array.*

Problem 7.50 *Can you devise a better fitness function for searching for Costas arrays? Find a function that is different from the one used in the chapter and document its strengths and weaknesses.*

Problem 7.51 *A tie in a random key encoding happens when two numbers in the array are equal. Does the use of non-aligned crossover increase the probability of ties within members of a population of random key encodings? If the answer is yes, give a technique for eliminating such ties.*

Problem 7.52 *For each of the following permutations in one-line notation, compute the number of violations of the Costas condition. Hint: using a computer may be faster and is almost surely more accurate.*

(i) **5 1 7 4 3 9 6 8 2 0**

(ii) **0 3 6 2 1 9 7 8 4 5**

(iii) **8 7 4 2 9 1 3 5 0 6**

(iv) **2 7 1 8 6 5 9 0 3 4**

(v) **9 4 7 8 1 2 6 3 5 0**

(vi) **7 9 3 5 0 4 8 6 1 2**

(vii) **5 0 7 1 3 2 8 6 4 9**

(viii) **8 5 0 4 7 6 1 3 9 2**

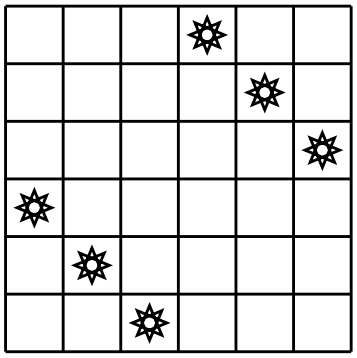
(ix) **3 7 2 1 8 9 6 4 0 5**

(x) **6 8 4 3 1 2 0 7 5 9**

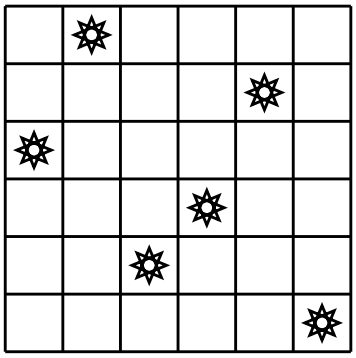
(xi) **6 7 0 5 1 4 2 8 9 3**

(xii) **2 7 4 3 0 6 5 1 9 8**

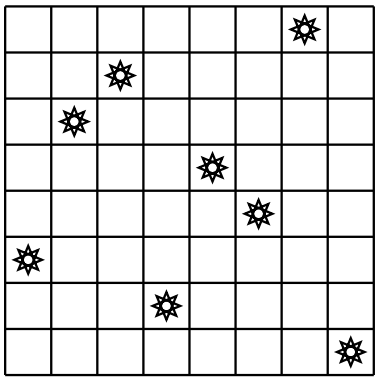
Problem 7.53 *For each of the following 4 arrays, compute the number of violations of the Costas condition and, on a copy of the matrix presentation, show the vectors that repeat more than once.*



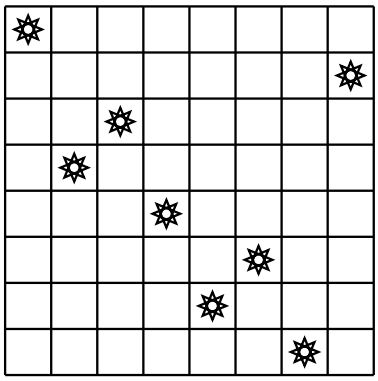
(i)



(ii)



(iii)



(iv)

Problem 7.54 *Prove Theorem 7.6.*

Problem 7.55 *A fixed point in a permutation σ is a number for which $\sigma(i) = i$. Prove the strongest upper bound you can on the number of fixed points, if $VCC(\sigma) = 0$ (i.e., if M_σ is a Costas array).*

Problem 7.56 *Write a program that exhaustively enumerates all Costas arrays of a given order, and, either using your code or working with pencil and paper, compute the largest number of fixed points in Costas arrays of order n, for $n = 1, 2, \ldots, 10$.*

Problem 7.57 *Invent and code a deterministic greedy algorithm for placing the successive dots of a permutation matrix so as to minimize the number of violations of the Costas condition. Make your algorithm such that it can find Costas arrays of all orders up to 5.*

Problem 7.58 *Compute the number of permutation matrices of order n that are symmetric about their main diagonal. Hint: such permutation matrices correspond to permutations of order 2. Compute also the fraction of all permutation matrices that are symmetric in this fashion.*

Problem 7.59 Essay. *Address the following question: would it be a good idea to search for Costas arrays that are symmetric about their main diagonal?*