Greedy, Genetic, and Greedy Genetic Algorithms for the Quadratic Knapsack Problem

Bryant A. Julstrom
Department of Computer Science
St. Cloud State University
St. Cloud, MN, 56301 USA
julstrom@stcloudstate.edu

ABSTRACT

Augmenting an evolutionary algorithm with knowledge of its target problem can yield a more effective algorithm, as this presentation illustrates. The Quadratic Knapsack Problem extends the familiar Knapsack Problem by assigning values not only to individual objects but also to pairs of objects. In these problems, an object's value density is the sum of the values associated with it divided by its weight. Two greedy heuristics for the quadratic problem examine objects for inclusion in the knapsack in descending order of their value densities. Two genetic algorithms encode candidate selections of objects as binary strings and generate only strings whose selections of objects have total weight no more than the knapsack's capacity. One GA is naive; its operators apply no information about the values associated with objects. The second extends the naive GA with greedy techniques from the non-evolutionary heuristics. Its operators examine objects for inclusion in the knapsack in orders determined by tournaments based on objects' value densities. All four algorithms are tested on twenty problem instances whose optimum knapsack values are known. The greedy heuristics do well, as does the naive GA, but the greedy GA exhibits the best performance. In repeated trials on the test instances, it identifies optimum solutions more than nine times out of every ten.

Categories and Subject Descriptors

G.1.6 [**Mathematics of Computing**]: Numerical Analysis—*Optimization*; G.2.1 [**Mathematics of Computing**]: Discrete Mathematics—*Combinatorics*; I.2.6 [**Computing Methodologies**]: Artificial Intelligence—*Learning*

General Terms

Algorithms

Keywords

Quadratic Knapsack Problem, genetic algorithms, greedy operators

GECCO'05, June 25–29, 2005, Washington, DC, USA.

1. INTRODUCTION

Augmenting an evolutionary algorithm with knowledge of its target problem often yields an algorithm that identifies better solutions. We illustrate this general observation with greedy, genetic, and greedy genetic algorithms for the Quadratic Knapsack Problem. This problem extends the familiar Knapsack Problem by assigning values not only to individual objects that might go in the knapsack but also to pairs of objects. When the knapsack holds both objects of a pair, the pair's value is added to the knapsack's total. Section 2 below describes the problem in more detail, Section 3 describes two related problems, and Section 4 describes twenty test instances for which optimum knapsack values are known.

In the Quadratic Knapsack Problem, an object's value density is the sum of all the values associated with it divided by its weight. Two greedy heuristics, which Section 5 describes, try to fit objects into the knapsack in decreasing order of their value densities. One computes each object's value density with respect to all the other objects in the problem instance, the other with respect to the objects in the knapsack so far. The latter algorithm returns better solutions on the test instances.

Two genetic algorithms represent candidate selections of objects for the knapsack with binary strings, and both generate and manipulate only strings whose selections of objects do not exceed the knapsack's capacity. A naive GA, which Section 6 presents, applies operators that do not use information about objects' values. It returns good but uneven results on the test instances.

A greedy GA, which Section 7 presents, augments the operators of the naive GA with greedy techniques from the non-evolutionary heuristics. In particular, it chooses objects in tournaments based on their value densities when it generates the chromosomes that make up its initial population, in crossover, and in mutation. The resulting algorithm is very effective, identifying optimum solutions on more than 90% of repeated trials on the test instances. On only one instance is its performance worse than that of the naive GA.

While researchers have described evolutionary algorithms for related problems, the GAs described here are, to my knowledge, the first for the Quadratic Knapsack Problem.

2. THE PROBLEM

In the familiar Knapsack Problem (KP), we are given a collection of n objects, each with a positive value v_i and a positive weight w_i, and a knapsack with capacity C. The

goal is to select objects to put in the knapsack whose total value is as large as possible but whose total weight does not exceed C. More formally, let x_1, x_2, ..., x_n be binary variables that indicate the selection ($x_i = 1$) or exclusion ($x_i = 0$) of each object. We seek to assign values to these variables, indicating object choices, to maximize

$$V = \sum_{i=1}^{n} x_i v_i$$

while maintaining

$$W = \sum_{i=1}^{n} x_i w_i \leq C.$$

KP is an archetypal problem of constrained combinatorial optimization. It is NP-hard [9, p.65]. A simple greedy heuristic for it sorts the objects by their "value densities:" the ratios v_i/w_i of their values to their weights, scans the objects in descending order of these ratios, including all those that fit in the knapsack, then returns either the selected objects or the one object of largest value, whichever has the largest total value. The objects this algorithm selects always have total value at least half that of an optimum selection.

The Quadratic Knapsack Problem (QKP) extends the Knapsack Problem by introducing quadratic terms into the value computation. In QKP, in addition to the value v_i associated with each object alone, there is a non-negative value v_{ij} that accrues when the knapsack contains both object i *and* object j. Thus,

$$V = \sum_{i=1}^{n} x_i v_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} x_i x_j v_{ij}. \quad (1)$$

The weight computation and the capacity constraint are unchanged from KP. QKP is NP-hard by restriction to KP; set all the quadratic values v_{ij} to zero.

The Quadratic Knapsack Problem was first described by Gallo, Hammer, and Simeone [8], who presented a branch-and-bound algorithm for it. Exact algorithms for QKP have applied branch-and-bound [4] [11], Lagrangian relaxation [1] [4], semidefinite programming [12], mixed-integer programming [3], and other techniques. Kellerer, Pferschy, and Pisinger [15, Ch.12] present a thorough introduction to QKP and exact and approximate algorithms for it.

QKP arises in a variety of applications, including finance [17], VLSI design [7], and location problems [24]. Also, various problems in graphs can be presented as instances of QKP [13] [21].

3. RELATED PROBLEMS

QKP is related to Binary Quadratic Programming (BQP), also called Unconstrained 0-1 Quadratic Programming, in which the values v_i and v_{ij} may be positive, zero, or negative and there are no weights or capacity constraint. Another generalization of the 0-1 Knapsack Problem is the Multidimensional Knapsack Problem, also called the Multiconstrained Knapsack Problem. In this problem, each object has a value v_i and a resource demand s_{ik} for each of m resources. Of each of these resources k there is a fixed supply S_k, and the goal is to choose objects of maximum total value without demanding more of each resource than is available:

$$\sum_{i=1}^{n} x_i s_{ik} \leq S_k, k = 1, \ldots, m.$$

Researchers have described a variety of evolutionary algorithms for Binary Quadratic Programming [14] [18] [19] [20] and for the Multidimensional Knapsack Problem [5] [10] [16] [22] [23].

4. SOME QKP INSTANCES

An instance of the Quadratic Knapsack Problem consists of its number n of available objects, linear object values v_i, quadratic object values v_{ij}, object weights w_i, and knapsack capacity C. A significant feature of a QKP instance, not to be confused with the value densities of its objects, is the proportion of its quadratic values v_{ij} that are non-zero. This proportion is called the density of the instance, and it indicates the level of interaction between the instance's objects. The lower an instance's density, the more independent are objects' contributions to a knapsack's value; the higher its density, the more objects' contributions depend on the other objects in the knapsack.

Another feature of a QKP instance is the ratio $C/\sum w_i$ of the instance's capacity to the sum of all its objects' weights. This ratio corresponds roughly to the proportion of the objects likely to appear in an optimum solution.

Billionnet and Soutif, who have written extensively on the Quadratic Knapsack Problem, provide an on-line collection of randomly generated QKP instances[1]. The algorithms that the next three sections describe were tested on twenty of these instances. Ten of the instances consist of $n = 100$ objects. These instances have density 0.25 and ratios $C/\sum w_i$ ranging from 0.06 to 0.93. The remaining ten instances consist of $n = 200$ objects. These have density 1.00 and ratios $C/\sum w_i$ ranging from 0.03 to 0.93. All have been solved to optimality [2] [3], and the web site reports their optimum values. The left half of Table 1 summarizes the instances.

5. GREEDY HEURISTICS

The genetic algorithm of Section 7 applies greedy variation operators. This section presents two simple greedy heuristics on which those operators are based and reports on their performance on the test QKP instances.

5.1 Absolute Greed

An obvious greedy heuristic for the Quadratic Knapsack Problem imitates the heuristic for the simple Knapsack Problem given in Section 2. Let the absolute value density d_i of an object i in a QKP instance be the ratio of the sum of all the values associated with the object to its weight:

$$d_i = \frac{v_i + \sum_{j \neq i} v_{ij}}{w_i}. \quad (2)$$

Compute the value densities of all the instance's objects and sort the objects into descending order of these values. Then scan the objects in their sorted order. Include in the solution all the objects that fit into the knapsack; that is, whose inclusion does not cause the sum of the included objects'

[1] `http://cermsem.univ-paris1.fr/soutif/QKP/QKP.html`

weights to exceed C. The following sketch summarizes this simple greedy heuristic; S is the solution it builds.

```
compute objects' value densities;
sort the objects by their value densities;
S ← ∅;
weight ← 0;
for the objects in sorted order
  i ← next object;
  if ( weight + w_i ≤ C )
    S = S ∪ {i};
    weight ← weight + w_i;
evaluate and report S;
```

Computing the value density of one object requires time that is $O(n)$, so finding all the value densities is $O(n^2)$. Sorting the objects is $O(n \log n)$. Scanning the objects requires only linear time, and evaluating the resulting solution is $O(n^2)$, so the entire algorithm requires time that is $O(n^2)$.

5.2 Relative Greed

The value densities d_i are called absolute because each is computed with respect to all the objects in a QKP instance, regardless of whether or not they appear in a particular solution. A second heuristic applies value densities based on the objects currently in a growing solution.

Let the relative value density r_i of an un-included object i with respect to a (partial) solution S be

$$r_i = \frac{v_i + \sum_{j \in S} v_{ij}}{w_i}; \tag{3}$$

that is, the numerator sums only the quadratic values associated with i and the objects currently in S. The second heuristic begins with each object in turn and repeats the following steps. It computes the relative value densities of the un-included objects with respect to the included ones and adds to the solution the object i that fits and has the largest r_i, continuing until no more objects can be added to the solution. It then reports a solution with the largest total value.

Call this algorithm the relative greedy heuristic. In the following sketch, T is each candidate solution, and S is the best solution so far.

```
v ← 0;
for each object i from 1 to n
  T ← {i};
  repeat
    for each object j ∉ S
      update r_j with respect to S;
    add to T the legal object j with largest r_j;
  until no more objects can be added;
  v_T ← value of T;
  if v_T > v
    S ← T;
    v ← v_T;
report S and v;
```

Within the innermost loop, updating the excluded objects' relative value densities requires only linear time: For each excluded object j, add v_{ij}/w_j to r_j, where i is the object just included. This step is repeated no more than n times, so the construction of one solution T is $O(n^2)$. This construction is repeated for each object as the first object, so the entire algorithm's time is $O(n^3)$.

5.3 Tests

The heuristics just described and the genetic algorithms below were implemented in C++ and executed on a Pentium 4 with 256 megabytes of memory running at 2.53 GHz under Red Hat Linux 9.0.

Table 1 presents the results obtained when the two heuristics were run on the twenty QKP instances of Section 4. For each heuristic on each instance, the table gives the value of the solution the heuristic identified, the percentage by which that value falls short of the known maximum value, and the time in seconds that the algorithm required.

The simple greedy heuristic is fast but not always effective. On the 100-object instances with density 0.25, it found an optimum solution on one, but missed two by about 4% and another by over 11%. On average, the values of its solutions fell short of the optimum values by about 2.9%. On the 200-object instances with density 1.0, it performed better. It again found one optimum solution, and on average fell short of the optima by only 0.7%. On every instance but two, it selected objects whose total values were within 1% of their instances' optima.

The relative greedy heuristic took much longer but performed much better. On the 100-object instances, it found an optimum solution five times. All the remaining solutions were within 1% of their respective optima, and the average error was less than 0.2%. On the 200-object instances, it found one optimum, every solution was within 0.5% of optimum, and its average error was again less than 0.2%.

6. A GENETIC ALGORITHM

This section presents a straightforward genetic algorithm for the Quadratic Knapsack Problem. This GA is naive in that its operators do not use objects' absolute or relative value densities when they construct chromosomes.

6.1 Coding

Binary strings of length n encode candidate solutions—selections of objects for the knapsack—as described, for example, by Lodi, Allemand, and Liebling [18] and Merz and Freisleben [19] for Binary Quadratic Programming and by Chu and Beasley [5] and Raidl [22] for the Multidimensional Knapsack Problem: In a chromosome `c[·]`, `c[`i`]` $= 1$ indicates the inclusion of object i; `c[`i`]` $= 0$ indicates its exclusion.

The GA handles the capacity constraint C by never generating chromosomes whose solutions violate it; no repair of penalty mechanism is necessary. To generate a random chromosome, for example, the GA considers the objects in random order and includes those whose presence in the chromosome's solution do not cause its total weight to exceed C, following the example of Raidl [22].

A chromosome's fitness,which the GA seeks to maximize, is the total value (1) of the solution it represents.

6.2 Variation Operators

The GA applies two variation operators that generate novel chromosomes from existing ones. These operators always build valid chromosomes.

Crossover builds one offspring from two parent chromosomes. It begins by including in the offspring all the objects that are common to the parents. It then scans the remaining parental objects in random order, adding to the offspring

Table 1: Performance of the simple greedy heuristic HEU and the relative greedy heuristic R-HEU on the twenty QKP instances. For each instance, the table lists its number n of objects, number, optimum value, and ratio $C/\sum w_i$. For each heuristic on each instance, it lists the value of the solution the heuristic identified, the percentage %E by which it fell short of the optimum, and the time in second that the algorithm ran.

Instance				HEU			R-HEU		
n	Num	Opt	$C/\sum w_i$	Value	%E	Time	Value	%E	Time
100	1	18558	0.26	17903	-3.53	< 0.01	18546	-0.06	0.33
	2	56525	0.85	56339	-0.33	< 0.01	56525	0.00	0.45
	3	3752	0.06	3326	-11.35	< 0.01	3717	-0.93	0.15
	4	50382	0.76	50157	-0.45	< 0.01	50382	0.00	0.44
	5	61494	0.93	61494	0.00	< 0.01	61494	0.00	0.45
	6	36360	0.54	35941	-1.15	< 0.01	36360	0.00	0.40
	7	14657	0.19	14049	-4.15	< 0.01	14545	-0.76	0.30
	8	20452	0.26	19651	-3.92	< 0.01	20452	0.00	0.35
	9	35438	0.57	34958	-1.35	< 0.01	35370	-0.19	0.40
	10	24930	0.40	24287	-2.58	< 0.01	24926	-0.02	0.37
200	1	937149	0.93	935700	-0.15	< 0.01	935700	-0.15	6.76
	2	303058	0.27	300313	-0.91	< 0.01	302644	-0.14	5.29
	3	29367	0.03	28455	-3.11	< 0.01	29222	-0.49	1.76
	4	100838	0.10	99610	-1.22	< 0.01	100838	0.00	3.39
	5	786635	0.78	785441	-0.15	< 0.01	786051	-0.07	6.65
	6	41171	0.04	41171	0.00	< 0.01	41171	0.00	2.12
	7	701094	0.69	698566	-0.36	< 0.01	699795	-0.19	6.56
	8	782443	0.76	779605	-0.36	< 0.01	780064	-0.30	6.66
	9	628992	0.63	626668	-0.37	0.01	628000	-0.16	6.45
	10	378442	0.36	376219	-0.59	< 0.01	377138	-0.34	5.68

those whose inclusion does not cause the offspring's solution to violate the capacity constraint.

Mutation builds one offspring from its one parent. It begins by building two complementary lists, one of the objects the parent includes, the other of the objects it excludes. With a probability based on the number of included objects, it may remove each of them from the offspring. It then, as in crossover, scans the excluded objects in random order and includes in the offspring all that fit. While crossover, which examines only parental objects, may leave room in offspring for additional objects and therefore for greater total value, mutation examines all the excluded objects and thereby fills its offspring to capacity.

6.3 The GA

A generational genetic algorithm applies these operators. The GA initializes its population with random chromosomes and selects chromosomes to be parents in k-tournaments without replacement. It applies crossover and mutation independently; each offspring is generated by one operator or the other, never both. It is 1-elitist, preserving the best chromosome of the current generation unchanged into the next, and it runs through a fixed number of generations.

Crossover could have been followed by mutation to produce each offspring, but the present organization allows convenient control of the operators' probabilities. In any case, the interactions of an EA's representation with its operators are more important in determining its performance than is the organization of its operators' application.

In the tests the next section describes, when run on a QKP instance with n objects, the GA's population contained n chromosomes. The size of its selection tournaments was two, and each tournament's winner was in fact selected to be a parent with probability 0.90, its loser therefore with probability 0.10. The probability that crossover generated each offspring chromosome was 0.70, the probability of mutation therefore 0.30, and in mutation, the probability that each parental object was excluded from the offspring was $2/I$, where I was the number of objects in the parental solution. The GA ran through $10n$ generations.

6.4 Tests

The genetic algorithm just described was run 50 independent times on each of the twenty QKP instances. Table 2 summarizes the results of these trials. For each instance, it lists the number of trials that identified an optimum solution, the value of the best solution found, the average of the trials' best solutions, the average percentage by which the GA's solutions fell short of the optimum, the standard deviation of the GA's solutions, and, for the trials that found an optimum solution, the minimum and mean numbers of generations and the mimimum and mean times in seconds that this required. The table also reports the total number of trials that found optimum solutions out of the 500 trials on each set of instances.

As Table 2 shows, the GA's performance varies widely over the test QKP instances. It quickly finds optimum solutions on two of the 100-object instances, and 48 times out of 50 on a third, but on the last two 100-object instances, it almost never hits. Overall, on the 100-object instances, it finds optima on 281 trials out of 500, and its average error is just under 0.2%. On all the instances but the ninth, the shortest time the GA required to find an optimum solution was less than the greedy heuristic's running time, and on two instances, the GA's average time to an optimum solution was less than the heuristic's time.

Table 2: Results of 50 trials of the naive genetic algorithm on each of the test QKP instances. For each instance, the table lists the number of trials on which the GA found an optimum solution (Hits) the maximum and mean values of the solutions the GA found, the mean percent by which the GA's solutions fell short of the optimum, the standard deviation of the 50 values, and the minimum and mean number of generations and minimum and mean time in seconds that the GA used to find an optimum solution, when it did.

Instance					Mean		Gens to Opt		Time to Opt	
n	Num	Hits	Max	Mean	%E	StdDev	Min	Mean	Min	Mean
100	1	18	18558	18527.2	-0.17	37.58	38	286.28	0.24	4.50
	2	50	56525	56525.0	0.00	0.00	12	40.78	0.08	0.25
	3	26	3752	3734.9	-0.46	18.10	16	117.85	0.10	3.29
	4	31	50382	50370.8	-0.02	18.67	28	117.65	0.18	2.73
	5	50	61494	61494.0	0.00	0.00	5	13.66	0.04	0.09
	6	15	36360	36217.6	-0.39	98.05	39	549.33	0.25	5.23
	7	37	14657	14632.0	-0.17	46.33	21	182.62	0.14	2.40
	8	48	20452	20447.2	-0.02	24.21	21	107.90	0.14	0.88
	9	2	35438	35298.1	-0.39	112.51	707	720.00	4.29	5.98
	10	4	24930	24894.2	-0.14	52.02	43	162.25	0.27	5.65
Total hits: 281/500										
200	1	50	937149	937149.0	0.00	0.00	20	43.88	0.98	2.08
	2	37	303058	303020.8	-0.01	69.74	48	176.57	2.31	30.31
	3	48	29367	29364.2	-0.01	14.05	18	39.83	0.89	5.55
	4	50	100838	100838.0	0.00	0.00	26	47.08	1.27	2.25
	5	21	786635	786102.7	-0.07	1531.89	45	658.71	2.14	66.60
	6	49	41171	41149.4	-0.05	152.88	16	21.33	0.80	2.89
	7	26	701094	701010.2	-0.01	112.73	49	239.42	2.34	50.34
	8	30	782443	782415.9	-0.00	44.46	46	240.47	2.19	43.78
	9	15	628992	628572.1	-0.07	947.53	54	349.60	2.59	69.87
	10	35	378442	378410.8	-0.01	56.32	54	264.83	2.59	36.58
Total hits: 361/500										

On the 200-object instances, the GA's performance is better. On two of these instances, it quickly finds an optimum solution on every trial, and 49 times out of 50 on a third. On no instance does it find an optimum solution fewer than fifteen times. Overall, it hits on 361 trials out of 500, and its average mean error is only 0.023%. The GA's shortest time to an optimum solution is always less than the greedy heuristic's running time, as was the GA's average time to an optimum solution on two of the 200-object instances.

7. WITH GREEDY OPERATORS

The performance of the genetic algorithm just described can be improved by introducing into it greedy techniques from the algorithms of Section 5. The resulting greedy GA favors objects with higher value densities when it builds random chromosomes, in crossover, and in mutation. In all three operators, this strategy depends on a parameter called T_BIAS, chosen on the interval $(0.5, 1.0]$.

7.1 Operators

As in the naive GA, all the operators generate only chromosomes whose solutions satisfy the capacity constraint.

To build chromosomes for its initial population, the greedy GA begins by computing the absolute value densities (2) of all the objects. To build each chromosome, the GA considers the objects one at a time. The object to consider next is determined by a tournament with replacement of two objects randomly chosen from those not previously considered. With probability T_BIAS, the object with larger value density is considered next; with probability $(1 - \text{T\_BIAS})$, it's the object with smaller value density. If this object fits in the knapsack, it is included; otherwise not. In either case, that object cannot be considered again, though its competitor in the tournament can and will. This process continues until all the objects have been considered. (Initialization could compare objects' relative value densities (3) for all objects after the first, but this strategy was found to lead the GA to inferior solutions.)

Crossover begins, as in the naive GA, by initializing its offspring with the objects common to both parents. It next computes the relative value densities (3) for the remaining parental objects with respect to the common objects. It completes the offspring by considering the remaining objects in an order determined by 2-tournaments with parameter T_BIAS, as in initialization but based on the objects' relative value densities, which it updates after an object is included in the offspring.

Mutation also begins as in the naive GA. It makes two lists, of the parental objects and of the objects the parent solution does not contain, and it probabilistically removes some of the included objects. It computes the relative value densities of the excluded objects with respect to the remaining included ones and, as in crossover, scans the excluded objects in an order determined by 2-tournaments with parameter T_BIAS. These tournaments compare objects' relative value densities, which are updated after an object is added to the offspring. Again, crossover may not fill its offspring to capacity, but mutation does.

Table 3: Results of 50 trials of the greedy genetic algorithm on each of the test QKP instances. For each instance, the table lists the number of trials on which the GA found an optimum solution (Hits) the maximum and mean values of the solutions the GA found, the mean percent by which the GA's solutions fell short of the optimum, the standard deviation of the 50 values, and the minimum and mean number of generations and the minimum and mean time in seconds that the GA used to find an optimum solution, when it did.

Instance					Mean		Gens to Opt		Time to Opt	
n	Num	Hits	Max	Mean	%E	StdDev	Min	Mean	Min	Mean
100	1	50	18558	18558.0	0.00	0.00	7	44.88	0.08	0.37
	2	50	56525	56525.0	0.00	0.00	1	3.24	0.02	0.03
	3	36	3752	3742.2	-0.26	15.87	12	183.56	0.12	3.53
	4	23	50382	50368.5	-0.03	12.59	3	96.48	0.03	3.92
	5	50	61494	61494.0	0.00	0.00	0	0.58	< 0.01	0.01
	6	50	36360	36360.0	0.00	0.00	7	26.10	0.06	0.21
	7	50	14657	14657.0	0.00	0.00	5	9.22	0.05	0.09
	8	50	20452	20452.0	0.00	0.00	5	7.72	0.05	0.08
	9	37	35438	35419.4	-0.05	31.78	4	235.05	0.04	3.10
	10	50	24930	24930.0	0.00	0.00	6	11.14	0.07	0.10
Total hits: 446/500										
200	1	50	937149	937149.0	0.00	0.00	15	469.40	0.80	22.72
	2	50	303058	303058.0	0.00	0.00	29	103.12	1.95	6.22
	3	50	29367	29367.0	0.00	0.00	12	18.90	0.90	1.37
	4	50	100838	100838.0	0.00	0.00	12	20.32	0.92	1.47
	5	50	786635	786635.0	0.00	0.00	16	48.78	0.95	2.61
	6	50	41171	41171.0	0.00	0.00	11	13.42	0.83	1.01
	7	50	701094	701094.0	0.00	0.00	18	196.16	1.12	10.25
	8	6	782443	782398.1	-0.01	27.44	1079	1570.50	54.50	98.23
	9	50	628992	628992.0	0.00	0.00	15	65.66	0.98	3.65
	10	50	378442	378442.0	0.00	0.00	39	178.82	2.47	10.31
Total hits: 456/500										

7.2 The GA

Structurally, the greedy GA is identical to its naive predecessor. It is generational and 1-elitist; it applies crossover and mutation independently; and it runs through a fixed number of generations. With one exception, its parameters have the same values as well. On a QKP instance with n objects, its population contains n chromosomes; in selection, the probability that a tournament's winner is selected in 0.70 (rather than 0.90 as in the naive GA); the probability that crossover generates each offspring is 0.70; the probability that mutation removes each included object is $2/I$; and the GA runs through $10n$ generations.

The greedy GA also includes the parameter T_BIAS, which controls the favor given objects of higher value densities in initialization, crossover, and mutation. For the 100-object instances with $C/\sum w_i = 0.25$, T_BIAS is set to 1.00. For the 200-object instances with $C/\sum w_i = 1.00$, it is 0.75.

7.3 Tests

The greedy genetic algorithm was run 50 independent times on each of the twenty QKP instances. Table 3 summarizes the results of these trials, in the same format as Table 2.

As expected, the greedy GA was more effective than the naive algorithm. It found better solutions in general and optimum solutions more quickly. On the ten 100-object instances, every trial found an optimum solution on seven, and always in very few generations. On one instance, the minimum number of generations to an optimum solution was zero, indicating that greedy initialization alone was, at least once, enough to solve the instance. On the remaining three instances, the numbers of hits were 36, 23, and 37, and the mean errors were all much less than 1%. The total number of hits over all ten instances was 446 in the 500 trials. The times the GA required to find optimum solutions were smaller than those of the naive GA on nine of the ten instances, and its mean times to optimum solutions were less on eight instances, often by large margins.

On the ten 200-object instances, the greedy GA's performance was even better. On nine of the instances, it always found an optimum solution, and the average number of generations it required to do so was in general small, ranging from 13.42 to 469.40, with five of these averages below 100. The greedy GA found an optimum solution in 456 of the 500 trials on these instances. It was also fast. On seven instances, its shortest time to an optimum solution was less than that of the naive GA, as was its mean time to an optimum solution on eight instances. The greedy operators' improved performance more than makes up for the additional time that they require.

The eighth 200-object instance is an anomaly. Only on this instance did the greedy GA sometimes fail to find an optimum solution and only on this instance was the greedy GA's performance worse than that of the naive GA. Thirty of the 50 trials of the naive GA found an optimum solution to this instance, but only six of the 50 trials of the greedy GA. This suggests that the structure and values in this instance conspire to mislead greedy heuristics, even relatively gentle

ones such as the greedy operators with T_BIAS = 0.75. On the other hand, the average error of the greedy GA's trials on this instance was still just 0.01%.

8. DISCUSSION

The results above support the effectiveness of the heuristics implemented in the greedy GA, but they also raise questions about the test instances and the algorithms.

The non-evolutionary heuristics and both GAs perform well on the test instances, which suggests that the instances are easy and that they might be more economically addressed by a simple randomized hill-climber. The instances were randomly generated [2] [3] and therefore are not structured to challenge or mislead the heuristics. Drezner, Hahn, and Taillard [6] described instances of the quadratic assignment problem designed to mislead the heuristics commonly applied to that problem; it should be possible to similarly design QKP instances tht challenge the present heuristics. Also, test instances could simply be larger.

The probability that the winner of a selection tournament is in fact selected to be a parent was set to 0.90 in the naive GA, but to 0.70 in the greedy GA. When the genetic operators are naive, selection pressure can be higher, but when the operators greedily identify improved chromosomes, lower selection pressure, achieved by reducing the probability that a tournament's winner reproduces, helps the GA preserve diversity in its population and avoid convergence to inferior solutions.

In the greedy GA's initialization, crossover, and mutation, the parameter T_BIAS is the probability that, in a 2-tournament of candidate objects, the one with higher value density will be considered next. The GA is sensitive to this parameter, which was set to 1.00 for the 100-object instances, whose density—proportion of non-zero quadratic values—was 0.25, and to 0.75 on the 200-object instances, whose density was 1.00. These values suggest that when the density of the target instance is low—when a strong majority of its quadratic values are zero—operators can be aggressively greedy because interaction between object choices—the epistasis of the representation—is low. When the target instance's density is high, there is in general much interaction between object choices, and excessive favor to objects of high value density stumbles on this interaction.

Tables 2 and 3 list the minimum and mean times that the two GAs require to identify optimum solutions, when they do, to the test problem instances. In general, however, we do not know the optimum values of problem instances, and the algorithms must run through their assigned $10n$ generations. For the naive GA, this takes about 6.1 seconds on the 100-object instances and about 93 seconds on the 200-object instances. For the greedy GA, these times average about 7.3 and 113 seconds, respectively. The greedy GA's improved performance is worth the small additional time it requires.

9. CONCLUSION

The Quadratic Knapsack Problem extends the familiar Knapsack Problem by associating values both with individual objects and with pairs of objects. KP is then a restricted version of QKP in which the pairs' values are all zero.

Two simple greedy heuristics perform well on twenty instances of QKP. Both build solutions by choosing objects according to their value densities: the sum of their individual and paired values divided by their weights. The heuristic that computes value densities with respect to the objects chosen so far is both slower and returns better solutions than the heuristic that computes value densities with respect to all the objects.

A straightforward genetic algorithm for the problem encodes candidate solutions as binary strings and applies operators that always yield valid chromosomes. That is, the total weight of the objects they specify never exceeds the knapsack capacity. This GA produces good but varying results on the test instances. On average, the values of its solutions are never more than 0.5% below the optimum values, and they are often much closer, but the GA often fails to identify an optimum solution.

A greedy genetic algorithm applies operators that implement the strategies of the non-evolutionary heuristics. Its operators probabilistically favor objects of high value density; generating random chromosomes compares absolute value densities, while crossover and mutation compare relative value densities. This GA outperforms the naive one. It finds optimum solutions more often, and its average error is smaller. On only one test instance are its results inferior to those of the naive GA. These results illustrate the power of evolutionary algorithms augmented with heuristic strategies to achieve good results on combinatorial problems like the Quadratic Knapsack Problem.

10. REFERENCES

[1] A. Billionnet, A. Faye, and E. Soutif. An exact algorithm for the 0-1 quadratic knapsack problem. In *ISMP'97*, Lausanne, Switzerland, 1997.

[2] Alain Billionnet and Éric Soutif. An exact method based on Lagrangian decomposition for the 0-1 quadratic knapsack problem. *European Journal of Operational Research*, 157(3):565–575, 2004.

[3] Alain Billionnet and Éric Soutif. Using a mixed integer programming tool for solving the 0-1 quadratic knapsack problem. *INFORMS Journal on Computing*, 16(2):188–197, 2004.

[4] Alberto Caprara, David Pisinger, and Paolo Toth. Exact solution of the quadratic knapsack problem. *INFORMS Journal on Computing*, 11:125–137, 1999.

[5] P. C. Chu and J. B. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[6] Zvi Drezner, Peter M. Hahn, and Éric D. Taillard. A study of quadratic assignment problem instances that are difficult for meta-heuristic methods. Technical report, INA, Yverdon-les-bains, 2002. INA technical report.

[7] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey. Formulations and valid inequalities for node capacitated graph partitioning. *Mathematical Programming*, 74:247–266, 1996.

[8] G. Gallo, P. L. Hammer, and B. Simeone. Quadratic knapsack problems. *Mathematical Programming*, 12:132–149, 1980.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[10] Jens Gottlieb and Günther R. Raidl. Characterizing locality in decoder-based EAs for the Multidimensional Knapsack Problem. In Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, Edmund Ronald, and Marc Schoenauer, editors, *AE'99: Selected Papers form the 4th European Conference on Artificial Evolution*, pages 38–52, Berlin, 2000. Springer-Verlag.

[11] Peter L. Hammer and David J. Rader, Jr. Efficient methods for solving quadratic 0-1 knapsack problems. *INFOR*, 35(3):170–182, 1997.

[12] C. Helmberg, F. Rendl, and R. Weismantel. A semidefinite programming approach to the quadratic knapsack problem. *Journal of Combinatorial Optimization*, 4(2):197–215, 2000.

[13] E. L. Johnson, A. Mehrotra, and G. L. Nemhauser. Min-cut clustering. *Mathematical Programming*, 62:133–152, 1993.

[14] Kengo Katayama and Hiroyuki Narihisa. On fundamental design of parthenogenetic algorithm for the binary quadratic programming problem. In *Congress on Evolutionary Computation 2001 Proceedings*, pages 356–363. IEEE, May 2001.

[15] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, Berlin, 2004.

[16] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In Ed Deaton, Dave Oppenheim, Joseph Urban, and Hal Berghel, editors, *Applied Computing 1994: Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 188–193, New York, 1994. ACM Press.

[17] D. L. Laughhunn. Quadratic binary programming with applications to capital budgeting problems. *Operations Research*, 18:454–461, 1970.

[18] Andrea Lodi, Kim Allemand, and Thomas M. Liebling. An evolutionary heuristic for quadratic 0-1 programming. *European Journal of Operational Research*, 119:662–670, 1999.

[19] Peter Merz and Bernd Freisleben. Genetic algorithms for binary quadratic programming. In Wolfgang Banzhaf et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 417–424, San Francisco, CA, 1999. Morgan Kaufman.

[20] Peter Merz and Bernd Freisleben. Memetic algorithms for the unconstrained binary quadratic programming problem. *BioSystems*, 78(1–3):99–118, 2004.

[21] K. Park, K. Lee, and S. Park. An extended formulation approach to the edge-weighted maximal clique problem. *European Journal of Operational Research*, 95:671–682, 1996.

[22] Günther R. Raidl. An improved genetic algorithm for the Multiconstrained 0-1 Knapsack Problem. In *The 1998 IEEE International Conference on Evolutionary Computation Proceedings*, pages 207–211, Piscataway, NJ, May 1998. IEEE.

[23] Günther R. Raidl. Weight-codings in a genetic algorithm for the multiconstrained knapsack problem. In *Proceedings of the 1999 Congress on Evolutionary Computation CEC99*, pages 596–603, Piscataway, NJ, 1999. IEEE Press.

[24] J. Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17:200–207, 1970.