2nd Lt David Crow

ENG/20M

CSCE 686 Advanced Algorithms, Homework 7

­

*For all questions requesting an algorithm, give it in the form of the standard algorithm structure used by Talbi. Additionally, describe the CSCE 686 standard search elements in the comments.*

**Problem 1 – Talbi 3.3**

In genetic algorithms (GA), the crossover operator uses some characteristics of the two parents to generate an offspring. Suppose a distance measures the similarity between any two individuals. In the design of a crossover operator, to force an offspring to share genetic material from both parents, the following property has to be satisfied:

where and are the parents, is the generated offspring, and is the set of all possible offspring generated by the crossover operator .

Let us consider a problem with a binary encoding in which the Hamming distance is used. Which crossover operators among the 1-point crossover and the uniform crossover verify the property?

The 1-fold crossover satisfies this property. For two parents and of length with a distance , we can cross the parents at any location . In doing so, we obtain an offspring consisting of two segments, one of length and one of length . Without loss of generality, assume the segment comes from and the segment comes from . We now know that . Thus, we can confidently say the property is satisfied if . However, because is a segment of , is a segment of , and , we know that the segments cannot differ by more than the entire string. In other words, , and the property is satisfied.

Similarly, the uniform crossover satisfies this property, but we’ll use a different argument. For two parents and of length with a distance , we use the uniform crossover to build an offspring one character at a time. We can keep running totals of the distances between the offspring and its parents. Each time we select the -th character from one of the two parents, the distance count for the other parent increases by one if and only if . Because the two parents only differ by characters, neither of these running totals can reach anything larger than . For this reason, the property is satisfied.

We can relate this to the problem of maximally covering a grid with a set of UAVs. To do so, we can represent a grid as a binary string. Specifically, for a grid of size , we’ll need a binary string of characters. The -th character is if that grid square is not visited by a given UAV; it’s if the UAV does visit that square. It’s clear that we’ll need to follow this process for each UAV.

Regardless, we can use this binary representation to generate paths through the grid for a single UAV. Of course, each time we generate an offspring, we must verify that the path is actually viable; disjoint paths or those that do not start and end on edge squares are not viable candidates and should be removed from the population.

Still, we can obviously utilize genetic algorithms in our solution approach. We showed above that the property is satisfied for binary encodings using the 1-fold and crossover operators, so we can certainly encode our own problem as a set of binary strings to ensure we satisfy the property. It’s likely we can prove this property holds for some non-binary encodings, too.

In short, we believe genetic algorithms can assist in our search for a viable, successful algorithm to solve our specific NP-complete (or harder) problem(s).

**Problem 2**

Use a genetic algorithm (GA) for your project solution using one of the NP-complete problem objectives (or both); alternatively, do the same for the set covering problem (or a variation) or the maximal independent set/clique problem (or a variation). Use at least one heuristic. Don’t forget the PD/AD feasibility functions. Explicitly indicate your GA pseudo code (Talbi form, search elements). Discuss the “art” of GA parameter/function selection for your PD/AD. Reference at least one appropriate paper providing insight into your PD and this GA AD.

For extra credit, implement and test limited examples. Report in Barr/MF/Talbi form.

Because we’ve extensively covered the set covering problem (SCP) this quarter, we elect not to redefine the problem domain here. Additionally, the instructions for this problem do not require us to utilize the complete PD/AD process, so we generalize the process in standard prose below.

-----

For this problem, we’ll use a genetic algorithm to solve SCP. To do so, we will encode a potential solution to an SCP instance as a binary string. If our graph has nodes, then a string of length will suffice. The -th character is if the -th node is not in the selected set; the character is otherwise.

For a given individual, our fitness function evaluates the number of nodes covered and the size of the covering set. Because covering the entire set is more important than finding the minimal covering set, we prioritize the former. Once every individual in the population has a fitness value, we can probabilistically select parents for future iterations. In doing so, we see that our approach is akin to the roulette wheel selection method. We choose not to replace the parents in the population. For this reason, our population grows in size.

If we allow our population to grow until it contains every possible individual in the population, we’ve effectively completed an inefficient brute-force approach. For this reason, we generate individuals, where is the number of possible binary strings of length . Note that we do not selectively generate individuals – even if is already in the set, we might generate and append a new . We only filter out duplicate individuals when the genetic algorithm is finished.

In pseudocode, our algorithm looks like this:

GeneticClique(V, E):

let p1 be a set of zeros

let p2 be a set of ones

let population = [p1, p2]

i = 0

while i < count(vertices) \* count(vertices)

(new\_p1, new\_p2) = select\_parents(population)

population.push(generate\_successor(new\_p1, new\_p2))

i += 1

endwhile

population = filter\_duplicates(population)

solution = argmax(compute\_fitness, population)

return solution

select\_parents(population):

let choices =

for p in population:

choices.push(p, compute\_fitness(p))

endfor

p1 = weighted\_random\_select(choices)

do

p2 = weighted\_random\_select(choices)

while p1 == p2

return (p1, p2)

generate\_successor(p1, p2):

let r = random\_int(1, p1.length() – 1)

successor = slice(p1, 0, r) + slice(p1, r, p1.length())

return successor

compute\_fitness(p):

let n = p.length()

return covered\_set(p).length() + 1/n \* (vertices.length() – n)

We see that our algorithm is not overly complicated. Both and run in constant-time. We can also see that is dependent on the runtime of , but a hash function can give us constant-time complexity. Finally, is bounded by . Thus, our algorithm runs relatively quickly for moderately-sized . Note, however, that it does not guarantee optimality.

We might need to tune our fitness function. It’s possible that the current function (as written above) doesn’t effectively capture the necessary relationship between covered set size and covering set size. Additionally, we will probably need to tune our crossover so as to guarantee the feasibility of offspring. More investigation is required in this area.

**Extra Credit**

We’ve implemented this algorithm in Python on a 2017 MacBook Pro. For reference, the code is attached. The file “data.py” contains various problem instances; “main.py” contains the algorithm and all overhead.

We show below a table of results for some small, medium, and large problems.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Problem | Size | Vertices | Edges | Time (s) | Solution | Optimal? |
| 1 | Small | 5 | 8 | 0.004474 | 0 | Yes |
| 2 | Small | 7 | 9 | 0.027440 | 0, 5, 6 | Yes |
| 3 | Medium | 12 | 23 | 0.182867 | 0, 1, 8, 9 | No |
| 4 | Medium | 15 | 23 | 0.834459 | 4, 5, 6, 10, 11, 12 | No |
| 5 | Large | 21 | 45 | 6.829917 | 0, 1, 4, 5, 8, 9, 20, 21 | Yes |

Clearly, the execution time grows quickly as the size of the graph grows. This is directly caused by our population limited – we generate exactly individuals for a graph of vertices. This execution time increase, then, is expected.

We also see suboptimal performance, but this is also expected. In no way is the optimality of our algorithm guaranteed – this is an NP-complete problem, after all. However, we see that we do sometimes achieve an optimal solution, and our suboptimal solutions are, in both cases, no more than one or two nodes away from optimal.

Thus, it seems that this implementation is a useful SCP solver when optimality is not guaranteed (as is the case for all SCP solvers). Of course, fine-tuning our heuristic may allow for better performance. More investigation is required to answer whether this is true.

**References**

1. Talbi, El-Ghazali. “Metaheuristics.”
2. <https://en.wikipedia.org/wiki/Set_cover_problem>
3. <https://pdfs.semanticscholar.org/6881/7937ebf74d20c68e04ecc9b4aa04a4e8d8dc.pdf>
4. <https://github.com/jamescobonkerr/scp-genetic-algorithm>
5. <https://link.springer.com/article/10.1057/jors.1996.82>
6. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.4475&rep=rep1&type=pdf>
7. <https://www.sciencedirect.com/science/article/abs/pii/037722179500159X>

**Problem 3a – Talbi 3.8**

Given a population of individuals associated with their fitness values , a selection mechanism allows us to select individuals from the population . Compute the complexity of the -ary tournament selection, selection, selection, and the -fold binary tournament selection used in evolutionary programming.

Answers: , , ,

Ensure you understand the answers given; briefly discuss these answers.

The -ary tournament selection has a complexity of because we randomly select elements from and then conduct a tournament to identify the best element in the set of . Because we have to repeat this process times, we find the complexity to be .

The ) selection has a complexity of because we generate different offspring that each compete with the parents; if a given offspring is better than its parents, we add it to the population *and remove the parents*. In doing so, then, we restrict the size of our population by some amount, and we always leave the unselected individuals. This is where the comes in. The term, of course, exists because of the offspring generation. In total, then, our complexity is .

The selection has a complexity of because we generate different offspring that each compete with the parents; if a given offspring is better than its parents, we add it to the population. Thus, we’re looping through and then through . Without knowing whether or not , we can only say that our complexity is .

The -fold binary tournament selection has a complexity of because we loop over all individuals; in each loop iteration , we compare the -th individual to each of other individuals in a tournament. This gives us our point values that we can use to select the next parents. This last computation is constant-time, so the entire tournament process has a complexity of .

**Problem 3b – Talbi 3.9**

Given a genetic algorithm with a population of size , the fitness of the individuals is represented by the vector . The selection mechanism will select individuals . The time complexity of the proportional selection via roulette wheel is . Indeed, computing the sum can be performed with a complexity of . The following step is repeated times: generate a random number in the interval . The values must be sorted to find the index with . The stochastic uniform sampling (SUS) selection mechanism is a deterministic variant of the proportional selection via roulette wheel. It is well-known for its efficiency. Compute its time complexity.

Answer:

Ensure you understand the answer given; briefly discuss this answer.

Because SUS selects all parameters in one spin of the wheel, we don’t need to repeat the spinning process. We simply spin the wheel once and then calculate for our selected individuals. Because we already know that computing is , we know the entire process is also .