2nd Lt David Crow

ENG/20M

CSCE 686 Advanced Algorithms, Homework 8

­

*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 2.3**



A vertex cover of an undirected graph is a subset of the vertices of the graph containing at least one of the two end points of each edge:

The vertex cover problem (VCP) is the optimization problem of finding a vertex cover of minimum size in a graph. In the graph in Figure 2.45, is an example of a vertex cover. is another, smaller vertex cover. VCP is related to the independent set problem: is an independent set if and only if its complement, , is a vertex cover set. Hence, a graph with vertices has an independent set of size if and only if the graph has a vertex cover of size . Propose a representation and a neighboring structure for VCP.

One potential representation for a VCP instance is the binary string. For a graph , we can define a binary string of length , where if is in the vertex set and otherwise. Thus, we can represent the graphs in Figure 2.45 with the strings and , and we can find the binary string of some vertex cover with the function . A similar function gives us the vertex cover represented by some string .

We can say that the neighbor of a vertex cover is such that the Hamming distance between and is . Thus, any vertex cover can have a maximum of neighbors.

Can we apply the proposed solution to the independent set problem?

Yes, we can. By simply flipping every bit in a binary string , we find the complement of , and this complement is an independent set in . Of course, if is in the independent set, and otherwise. The vertex covers in Figure 2.45 give the two independent set strings and .

Alternatively, we could ignore the flipping function. Given a vertex cover binary string of the form , we can say that is an independent set in , where if is in the independent set, and otherwise. In doing so, we lose a small amount of interpretability, but we save a small amount of computation time.

Propose an incremental way to evaluate the neighbors.

Given a binary string of length , we can simply create copies of . We then iterate through the list of copies. For each copy , we simply flip the -th bit. In doing so, we easily generate every possible neighbor of . If we’re looking for a minimal VCP, we can simply repeat this process until the number of s in some binary string is minimized.

**Problem 2**

Briefly discuss how you would use the neighborhood of Problem 1 to solve the vertex covering problem using simulated annealing.

As discussed above, we can easily generate all neighbors of some vertex cover by using binary strings. If we maintain a list of vertex covers we’ve already visited (that is, an explored set), we can ensure we don’t generate duplicate vertex covers.

We can also evaluate a given vertex cover by simply counting the number of s in the string; if the number is relatively low, the cover is relatively good. In a standard local search, we simply generate every unique neighbor of every cover in our frontier and visit only the best cover(s) at each step.

To solve VCP with simulated annealing, we follow the same procedure, but we also define a cooling schedule and probabilistically visit the best cover(s) at each step. This is a simple modification to the standard local search, and it improves our chances of finding a global optimum (as opposed to a local optimum).

**Problem 3**

Describe the neighborhood possibilities for your project to be used in a local search. Relate to the fitness landscape.

A set of UAV paths represents a possible solution to my problem. For example, if we have three UAVs with battery capacity and a grid of size , one possible set of UAV paths is

. We can see that this set of paths covers six unique grid squares. All neighboring solutions differ from this solution by exactly one square. In other words, a neighboring solution can contain five of the six squares, it can contain those six squares and one more square (on just one UAV path), or it can replace one of those squares with a different, legal square. A square on a path is illegal if is not adjacent to at least one of the other squares on the path.

The fitness function is difficult. This makes sense, because a polynomial-time fitness function would imply that my problem is in P. Still, we can define a heuristic function that considers the number of unique squares visited and the risk value over all squares visited. Some sort of weighted sum (inside our heuristic) can help guide our search for the state with the maximum coverage/minimum risk. In no way can this fitness function guarantee an optimal solution, but no fitness function can for such a difficult problem.

**Problem 4 – Talbi 2.19**

Suppose we have to solve the vehicle routing problem (VRP) that is defined in Example 1.11. The transformation operator used moves a customer from route to route . The tabu list is represented by the moves attributes. Propose three representations that are increasing in terms of their severity.

Here are the three representations:

1. Short-term memory, which prohibits reverse moves for iterations (for some user-defined ). In effect, if moves from route to route at time , we can a) prevent from returning to until after time , or b) we can prevent from leaving until after time . This reduces cycling.
2. Long-term memory, which diversifies the search by penalizing frequently-performed moves. To quote [1], “this is done by adding to the routing cost of a penalty term equal to the product of three factors: 1) a factor measuring the past frequency of the move; 2) a factor measuring instance size (such as ); 3) a user-controlled scaling factor.” Cordeau continues to say that this is computationally-inexpensive. We note here that it is more computationally-expensive than representation one, so this representation has a greater severity.
3. Intensification, which is a way to accentuate the search in promising regions. Effectively, intensification conducts periodic route improvements (perhaps by combining similar moves into a smaller number of moves, by reordering moves to allow for a better order, or by eliminating redundant moves) to optimize the solution. Of course, route improvements are not trivial to compute, so this is the representation with the greatest severity.

These three representations are all described in greater detail in [1].

**Problem 5a – Talbi 3.10**

The most computationally-intensive part of an evolutionary algorithm is the evaluation of the objective function. One of the possible solutions seeks to avoid a redundant evaluation of the same individuals. Propose an implementation of this solution.

Answer: (1) When creating an offspring clone from a parent, copy the parent’s objective function value into the offspring. Mark the offspring as evaluated. (2) After the application of a genetic operator (e.g., mutation, crossover) to an individual, check to see whether the resulting individual is different from the original one. If so, mark the individual as unevaluated. (3) Evaluate only the individuals that are marked unevaluated.

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

This solution certainly limits the number of redundant evaluations. Specifically, it ensures that the algorithm does not re-evaluate those offspring that are identical to the parent – only those that differ. Those that are the same as the parent simply receive the same objective function value.

This is a step in the right direction. However, if we assume we are searching for some specific offspring or for some offspring that meets some condition, then we should really seek to limit *all* redundant individuals in the population. The suggested answer does not do this. To implement this new approach, we would need to discard those individuals that already exist in the population.

As it stands, the suggested solution certainly reduces computation time, but it does not guarantee (or even try to achieve) a population with (mostly) unique individuals. It is certainly possible that this objective function allows for many, many redundant individuals.

**Problem 5b – Talbi 3.12**

The question in this exercise is to see if the crossover operators can generate solutions containing all possible combinations of elements of their parents. For simplification reasons, let us suppose a binary representation and the following crossovers: 1-point crossover, -point crossover, uniform crossover, shuffle crossover, and random respectful recombination.

Hint: Consider the strings 0 and 1 for the parents. What is discrete (nonbinary) strings.

Answer 1: Yes for uniform, shuffle, and random respectful recombination

Answer 2: Yes for random respectful combination

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

First, we note that the hint and answers are written very poorly. The second sentence in the hint is not a coherent sentence, and it’s clear that answer two is simply a repeat of the last part of answer one. Why does this question need two answers, anyway?

Regardless, the answers say that the uniform, shuffle, and random respectful recombination crossover operators can generate all possible offspring. This makes sense.

* The 1-point crossover operator can generate all offspring by simply crossing the two parents at every possible crossing point. We might need multiple levels of offspring (multiple generations), but 1-point can eventually generate all possible offspring.
* The same is true for the n-point crossover, but, because we can cross more times at each step, we can generate all offspring in just one generation.
* The uniform crossover, like the n-point crossover, can generate all offspring in one generation.
* For the same reason, the shuffle crossover can generate all offspring.
* The random respectful recombination [2] operator can generate all offspring because we are using binary strings and because the crossover operator uses stochasticity. Given enough time, the operator can generate every possible offspring from the parents.

**Problem 6 – Talbi 3.29 (pseudocode formulation)**

The set covering problem (SCP) represents an important class of NP-hard combinatorial optimization problems with many applications in different domains: transportation networks, integer coefficients linear programming, assignment problems, simplification of Boolean expressions, and so on. SCP is generally defined by its Boolean covering matrix: where and . An element of the matrix if the row is covered by the column . SCP concerns choosing a subset of columns at a minimal cost in such a way that all rows are covered (at least one on each line). In other words, the SCP concerns minimizing such that

and

The constraints maintain the integrity of the system. In other words, they guarantee that each row is covered by at least one column. The parameter is defined as the set of columns that cover line ; is the set of lines covered by column . ,

Propose a greedy algorithm to solve SCP. Based on this greedy algorithm, design a solution construction strategy for the ant colony optimization algorithm. Once a covering has been established by an ant, an iterative function that eliminates the redundant columns (those that are useless (superfluous) since their elements are covered by other columns of the covering) is performed. This function iterates on the columns, beginning from the last one whose cost is most significant, to optimize the cost.

Here's the greedy algorithm:

GreedySCPSolver():

1 Let

2 Let

// **solution**: check whether all rows are covered

3 While :

// **set of candidates**: although this list is currently

// empty, we fill it with potential columns

4 Let

// **next state generator**: builds list of those columns

// that we might add to our selected set

5 For to :

// **feasibility**: we only care about the columns

// that add new rows to our covered set

6 If :

// **heuristics**: compute a value for the

// column under consideration

7

8 If :

9 Return “no solution!”

// **objective**: we want the state with maximum value

10 Sort in decreasing order of

11 Let

// **selection**: pick the state that maximizes value

12 Let

13 Let

14 Return

This algorithm is likely not the most efficient implementation possible, but it is certainly a greedy algorithm that solves SCP. Because the algorithm is greedy, it’s not guaranteed to find the optimal solution.

Essentially, the algorithm selects the column that covers the maximum number of unique, uncovered rows, where this number is weighted by the cost of the column. We repeat this process until every row is covered, until every column is selected, or until no more columns can cover new rows. The if-statement on line 6 ensures we don’t reconsider columns we’ve already considered, and the if-statement on line 8 ensures we reach a stopping condition.

Here's the ant colony algorithm:

AntColonySCPSolver():

1 For every square :

2

3 For to :

4 Place ant on a randomly chosen column

// **solution**: initially null, but this (and ) will

// represent the solution at the end

5 Let be the smallest cover found from beginning

6 Let

7 For to :

// **set of candidates**: we place ants and thus

// consider possible candidates at each step

// **next state generator**: additionally, this loop

// generates future states for the algorithm by

// building multiple covers

8 For to :

9 Build cover by applying this times:

10 Choose the next column with probability

where is the current column

11 For to :

// **heuristics**: we care about the smallest cover

12 Compute size of cover produced by

// **selection**: we always keep track of the best cover

// found so far

// **feasibility**: rule out those covers that don’t

// improve the solution values

13 If an improved cover is found:

14 Update and

15 For every edge :

// **objective**: guide the search toward the best

// possible cover using pheromones (exploit)

16 Update pheromone trails by applying the rule

where

,

,

17 For every edge :

18

19 Return

This is merely a version of the ant colony pseudocode that solves the traveling salesman problem in [3]. I modified the pseudocode so that it can solve SCP. I also described the standard search elements in comments.

**References**

1. Cordeau, Jean-François. “Tabu Search Heuristics for the Vehicle Routing Problem.”
2. http://www.tomaszgwiazda.com/RandomRX.htm
3. Bonabeau et al. “Swarm Intelligence.”