Group 8

The optimal database design problem - final report

Academic Year 2018/2019

Group members:
Martina Alutto
Michele Da Re
Claudio Del Sole
Fulvio Di Stefano
Federico Paolucci
Roberta Raineri
Angelo Russi

1 The main idea: chromosomes

Our algorithm for solving the optimal database design problem is based on a genetic-type metaheuristic, combined with a first-improvement local search. A candidate solution to the problem is formalized as an array of length $|Q|$ (where Q is the set of queries), such that the entry i of this array is the configuration c associated to the query q_i. If a query isn't associated to any configuration, then its entry contains 0, representing the "default configuration": a fictitious configuration with zero cost, zero memory occupation and zero gain, associable to any query. This array is the "chromosome" that will be manipulated by a genetic algorithm, and its entries are the "genes".
We consider a population of chromosomes, whose cardinality is a parameter set at 32. These chromosomes work together to analyze the solution space. In our code, they are represented by an object of the class Chromosome, also containing the memory used by that solution and the objective function value attained by it.

2 Initialization

Once an instance of the problem has been loaded, our algorithm begins by randomly creating a population of starting chromosomes. This is managed by the method **buildInitialSolution()** of the class LocalSearch. For each of them, the following algorithm is applied:

1. A configuration c is randomly selected;
2. If the activation of c would bring the solution to exceed the memory constraint, c isn't activated and the initialization is concluded;
3. Otherwise configuration c is activated and all the queries that could be served by c are associated to the selected configuration, unless they are already served by a different one (different from the default one);
4. This procedure is iterated, always choosing a new configuration c' between those not yet chosen. If memory is never exceeded, it stops when all possible configurations have been checked.

Once the initial population is built, a local search is applied to each of its chromosomes. This step is usually performed after the genetic algorithm, therefore will be examined later.

3 Genetic algorithm

Our genetic algorithm pairs chromosomes of our population, which will be the parents. They are selected randomly and without repetitions. Children are then generated using the Standard Crossover, according to the following steps:

1. Two random numbers are chosen as cut-points in the chromosomes;
2. The two middle sections are switched in order to generate two new children.

This procedure doubles the population, since each pair of parents generates two children. We note that, with this genetic algorithm, children will never fall into one kind of infeasibility: there will never be a query associated to more than one configuration. However, they could be memory-infeasible.

4 Threads and local search

We use threads to analyze independently and in parallel every child. This improved the speed of our algorithm. For each thread, three methods are called. The first one is called **reassamble()**. It is possible that a query is not served, although there is an active configuration that can be associated to it. This method, therefore, tries to associate each non-served query to an active configuration. If more then one is available, the one with the highest gain is chosen. The second method is called **refining()**. It is likely that, after applying the genetics, the children are memory-infeasible. Therefore, for each child configurations are sorted according to the sum of the gains of all associated queries. Starting from the worst, they are turned off until the solution isn't memory-infeasible anymore. Finally, a **local search** method is invoked. Formally, a neighbourhood of each candidate solution is built and explored. We define as a neighbour of a solution (which is an array of length $|Q|$ containing the configurations) a solution that differs from our current one for at most one entry. The local search performs then a first improvement policy using the following algorithm:

1. A random query q is selected;
2. q is disassociated from the current serving configuration. The resulting net gain, both in memory and in objective function value, is computed;
3. All the configurations that can be associated to q are randomly explored, starting from the default configuration;
4. As soon as a configuration c is found such that there is a net improvement in the objective function value and the solution is memory-feasible, then query q is associated to configuration c;
5. If no improvement is found, then the previous configuration is kept.
6. A taboo list is kept in order to avoid cycling. Query q is added to the taboo list, and the list is checked each time a new query is selected.
7. The algorithm stops after a number of iterations set at eight times the number of queries or when there are $|Q|$ consecutive iterations without changes.

We note that this procedure corresponds to consider a query q and switch one link to another. Finally, a refining operation is done: all queries are visited in order and, if they can be linked to an already active configuration such that there is an improvement in the objective function value, this association is performed and the previous corresponding configuration is turned off.
Once the local search has concluded, a selection is applied to the resulting population. We decided to select the 4 best parents, the 8 best children, 16

random children and 4 random new solutions. The latter are generated with the same algorithm used for the initial solutions. The chromosomes of this selection are the parents of the following generation. If for more than 50 iterations we have no improvement of the objective function, we restart from 32 completely different chromosomes, in order to explore another part of the solution space.

5 Performance

Our algorithm was able to find every optimum in the test instances with a known optimum, and to improve most of the benchmark values in the instances 13-20. Due to the fact that some randomness is included in the algorithm, the time that is takes to solve an instance varies, as we can see in the following table.

	Benchmark	Opt	Test 1		Test 2		Test 3		Test 4		Test 5		Average		Gap %
			Obj.Fun	Time	Obj.Fun	Time	Obj.Fun	Time	Obj.Fun	Time	Obj.Fun	Time	Obj.Fun	Time	
instance01.odbdp	5951	YES	5951	2,0	5951	4,0	5951	1,0	5951	5,0	5951	1,0	5951	2,6	0,00
instance02.odbdp	1513	YES	1513	1,0	1513	2,0	1513	7,0	1513	1,0	1513	1,0	1513	2,4	0,00
instance03.odbdp	7484	YES	7484	74,0	7484	14,0	7484	18,0	7484	6,0	7484	56,0	7484	33,6	0,00
instance04.odbdp	22010	YES	22010	11,0	22010	75,0	22010	69,0	22010	33,0	22010	2,0	22010	38,0	0,00
instance05.odbdp	13874	YES	13874	66,0	13874	98,0	13874	35,0	13874	141,0	13874	93,0	13874	86,6	0,00
instance06.odbdp	7003	YES	7003	3,0	7003	22,0	7003	58,0	7003	2,0	7003	27,0	7003	22,4	0,00
instance07.odbdp	73772	YES	73772	120,0	73772	60,0	73403	300,0	73772	147,0	73772	117,0	73698	148,8	-0,10
instance08.odbdp	32999	YES	32999	152,0	32954	300,0	32954	300,0	32954	300,0	32954	300,0	32963	270,4	-0,11
instance09.odbdp	1710	YES	1706	300,0	1706	300,0	1710	25,0	1710	232,0	1697	300,0	1706	231,4	-0,25
instance10.odbdp	1672	YES	1672	178,0	1672	110,0	1672	28,0	1662	300,0	1672	208,0	1670	164,8	-0,12
instance11.odbdp	10749	YES	10749	74,0	10749	24,0	10749	52,0	10749	95,0	10749	80,0	10749	65,0	0,00
instance12.odbdp	2287	YES	2239	300,0	2239	300,0	2239	300,0	2239	300,0	2287	89,0	2249	257,8	-1,68
instance13.odbdp	26938	NO	27332	300,0	27332	300,0	27332	300,0	27332	300,0	27332	300,0	27332	300,0	1,46
instance14.odbdp	29280	NO	29280	300,0	29280	300,0	29280	300,0	29280	300,0	29280	300,0	29280	300,0	0,00
instance15.odbdp	12351	NO	12356	300,0	12356	300,0	12356	300,0	12356	300,0	12356	300,0	12356	300,0	0,04
instance16.odbdp	14110	NO	14332	300,0	14332	300,0	14332	300,0	14332	300,0	14332	300,0	14332	300,0	1,57
instance17.odbdp	12218	NO	12218	300,0	12218	300,0	12218	300,0	12218	300,0	12013	300,0	12177	300,0	-0,34
instance18.odbdp	2081	NO	2124	300,0	2146	300,0	2146	300,0	2125	300,0	2146	300,0	2137	300,0	2,71
instance19.odbdp	4257	NO	4257	300,0	4189	300,0	4189	300,0	4257	300,0	4189	300,0	4216	300,0	-0,96
instance20.odbdp	3406	NO	3417	300,0	3406	300,0	3339	300,0	3417	300,0	3352	300,0	3386	300,0	-0,58

We have also performed an analysis of the parameters used by our algorithm, in order to determine the best choice for their values. We focused on the cardinality of the population, on how many parents and children are taken into the next generation, and on the number of generations we wait before moving to another part of the solution space. The results were highly affected by randomness and we had to make a choice based on average. Moreover, we tried a few different initialization methods in order to decide what worked best with the algorithm.