University of Algarve

Faculty of Sciences and technology

Masters in Informatics Engineering

Metaheuristics

Nome: Diogo Fonseca
Número: 79858

Assignment 3

For each algorithm and problem instance, 100 independent runs were ran using the script “scripts/run_multiple.sh” and can be found in “logs/**X**_uf**Y**.txt”, where **X** is the algorithm and **Y** is the size of the problem instance. The average and variance can be simply calculated using the script “scripts/avg.sh <name_of_log_file>”.

Algorithms Used

Two algorithms were used: Simulated Annealing, and a Genetic Algorithm.

Simulated Annealing

For the simulated annealing algorithm, the hyperparameters were:

- **Hamming distance** (the hamming distance used to generate neighbours for the current node)
- **Maximum function evaluations** (the maximum amount of function evaluations before the algorithm stops)
- **Initial temperature** (the initial temperature for the algorithm)
- **Cooling schedule** (the function dictating how the temperature changes through time/iterations)

(the cooling schedule has to be defined inside the code, in input_reader.cpp).

For the given hyperparameters, the following were chosen by either thinking what made sense for the problem, and trial and error.
Hamming distance: 1.
Maximum function evaluations: 10 000 000.
Initial temperature: 1000.
Cooling schedule: f(T) = T * 0.999.

The hamming distance was chosen so that the algorithm could focus on exploring, without a need of spending too much time exploiting tons of nodes for early iterations, since it would be worse for a limited number of function evaluations, plus the time tradeoff seems to not be worth it.

The chosen cooling schedule gives us a slow exponential decay, slow enough to have a lot of exploration, but converging towards 0 early enough to exploit the current solution (especially when starting at 1000).

Below a graph can be seen with the cooling schedule, with the temperature on the y-axis and the number of iterations on the x-axis.

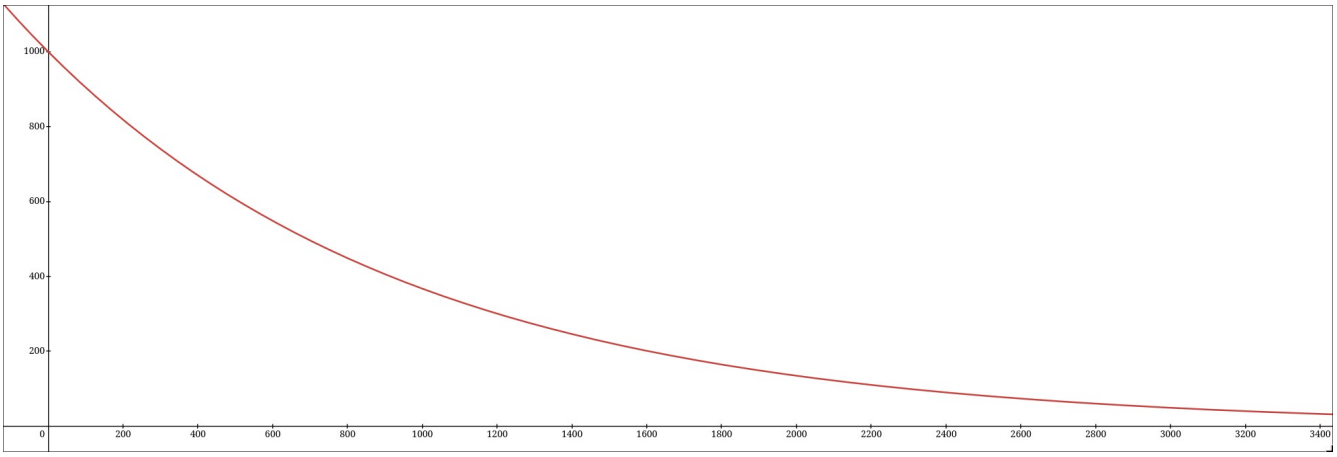

Figure 1: Visualization of the chosen cooling schedule with exponential decay. Temperature on the y-axis and iterations on the x-axis.

Genetic Algorithm

For initialization, the genetic algorithm generates random solutions. Then, for each generation, it will apply 3 operators:

1. **Tournament selection** (with no repetition regarding each match: Each match has n distinct individuals)
2. **k-point crossover** (crossover between 2 random distinct individuals at k points)
3. **Mutation** (a probability for each bit to flip)

Applied in that exact order, concluding one generation.

For the genetic algorithm, the hyperparameters were:

- **Maximum evaluations** (the maximum amount of function evaluations before the algorithm stops)
- **Population size** (the size of the population on each generation)
- **Number of crossover points** (the number of points to choose for crossover)
- **Mutation chance** (the chance for each bit to flip)

- **Tournament match size** (how many individuals participate in a tournament match)

For the given hyperparameters, the following were chosen by either thinking what made sense for the problem, and trial and error.
Maximum evaluations: 10 000 000.
Population size: 10000.
Number of crossover points: 5.
Mutation chance: 0.001.
Tournament match size: 3.

The hyperparameters were tuned so that it could find decent solutions within the maximum evaluations budget. If the population size was increased, for example, the number of evaluations would rise and the number of generations lower, leading to worse immediate solutions, so a balance was struck.

The number of crossover points chosen was 5, due to the 250-sized problem, but single crossover point was better for the 20 and 100-sized problems. This makes sense as we would be getting larger sections from each parent, which leads into better getting strings that satisfy the MAXSAT problem. Too many crossover points “muddy” the solution of the parent, making something completely different.

The mutation chance was chosen in such a way that it would preserve enough of the original individual, but tweaking it just enough to be able to possibly generate better solutions.

The tournament match size, if too high, is too elitist, if too low, it allows for more diversity, so a lower individual number per match was preferred. However, 3 was chosen over 2 since it eliminated some of the very bad solutions slightly quicker.

Results

Simulated Annealing

uf20-01.cnf

Metric	Average	Variance
CPU time	1.91ms	0.52ms
Function evaluations	13920.76	4130.79
Iterations	6960.38	2065.40
Fitness	91.00/91	0.00

uf100-01.cnf

Metric	Average	Variance
CPU time	621.58ms	679.34ms
Function evaluations	7611963.06	8276674.70
Iterations	3805981.53	4138337.35
Fitness	428.82/430	1.61

uf250-01.cnf

Metric	Average	Variance
CPU time	546.09ms	888.43ms
Function evaluations	5949244.68	9817371.04
Iterations	2974622.34	4908685.52
Fitness	1064.22/1065	1.66

Best runs

uf20-01.cnf	**Best quality solution obtained:** (91/91 clauses) 10010001010011101001 **Objective function evaluations:** 2680 **CPU time:** 0ms (not measurable by milliseconds, an insignificant amount of time).
uf100-01.cnf	**Best quality solution obtained:** (430/430 clauses) 00011001001000010110011111101111100110010011100011 00110011110001000110110111000010010111001000011011 **Objective function evaluations:** 19970 **CPU time:** 3ms.
uf250-01.cnf	**Best quality solution obtained:** (1065/1065 clauses) 00011010110101011001100011011011110100110000111100 00100100101101100110111100010001100001111111000110 00111100011100001001000110000100111011100000000001 11111110110111000111011101110111101010011111001111 01000011001111111010011010100110001001111110100100 **Objective function evaluations:** 38340 **CPU time:** 6ms

Genetic Algorithm

uf20-01.cnf

Metric	Average	Variance
CPU time	50.15ms	42.31ms
Function evaluations	250800.00	276959.82
Iterations (Generations)	8.36	9.23
Fitness	90.99/91	0.02

uf100-01.cnf

Metric	Average	Variance
CPU time	1603.63ms	100.13ms
Function evaluations	10020000.00	0.00
Iterations (Generations)	334.00	0.00
Fitness	426.53/430	2.31

uf250-01.cnf

Metric	Average	Variance
CPU time	4340.81ms	151.42ms
Function evaluations	10020000.00	0.00
Iterations (Generations)	334.00	0.00
Fitness	1060.57/1065	2.53

Best runs

uf20-01.cnf	**Best quality solution obtained:** (91/91 clauses) 10000100100001101001 **Objective function evaluations:** 30000 **CPU time:** 7ms.
uf100-01.cnf	**Best quality solution obtained:** (429/430 clauses) 00011001001001110111011101001000101100110110110011 10000111110001100110110110100110110111000000111001 **Objective function evaluations:** 10020000 **CPU time:** 1519ms
uf250-01.cnf	**Best quality solution obtained:** (1063/1065 clauses) 00111110111001011111100011111011000110110000111111 11101100101001000110111100001101101001111100000101 11110110110110001001010111000000111010100000101101 11001010111111100110011101011111100110101111001110 11100110001111011111011011001101010001001100000100 **Objective function evaluations:** 10020000 **CPU time:** 4215ms

Discussion

From the data obtained we can see that the simulated annealing algorithm is very efficient at the 20-sized problem, getting everything single instance correct in an insignificant amount of time. As the size of the problem instance grows larger, we can see that it converges to a solution extremely quickly. Furthermore, it's noteworthy that it has a smaller average time for the 250-sized problem than for the 100-sized problem. This was because the cooling schedule and temperature were tuned towards the 250-sized problem, and unchanged so that the measurements would be uniform. For the 100-sized problem, a lower initial temperature, and faster cooling schedule would make more sense, it spends too much time exploring. Furthermore, we note that the variance for this algorithm is **huge**, the bigger the size of the problem, the higher the variance. This algorithm could clearly benefit from an early stop (lower the number of max iterations) and multistart each time.

The genetic algorithm doesn't seem to perform that well for this problem, even having some (rare) instances where it doesn't solve the 20-sized problem.
It then always hit the maximum amount of function evaluations before reaching a solution, but so that a comparison could be made, this number was not increased. It only did 334 generations each time, which intuitively not enough for a genetic algorithm (especially with a population of 10000) to get to a good solution, however the times were still low, 1.6 and 4.3 seconds are very reasonable and competitive for the performance gotten.

From the results, we can clearly see with no further analysis needed that the simulated annealing completely outclasses this specific genetic algorithm, it's worst case consistently beating the genetic algorithm's best ones at a fraction of the time.

Conclusion

Simulated annealing seems to be a great algorithm to solve the MAXSAT problem. Furthermore, if paired with an early stop and multistart, it could solve the problem instances very efficiently.
As for the specific genetic algorithm implemented, it gets completely outclassed, although with pretty good solutions at a reasonable time frame.
Perhaps the genetic algorithm could have benefitted from different operators to the chromosomes/solutions. As the hyperparameters were fine tuned, it just means that for a better performance with the same amount of time, we need different operators.