```
PROCEDURE genetic_algorithm

  input
  n:                          number of individuals (i.e., solutions) in each population
  n_it:                       maximal number of iterations (i.e., successive populations)
  n_pairs_selected_parents:   number of parents to be selected in current population
  prob_mutation:              probability that a child be subject to a mutation
  k:                          maximal number of distinct best individuals in the last
                              population

  output
  best_solutions:             a set of at most k distinct solutions

pop ← initialize_population(n) ; i ← 1

while (not termination_condition){
 pairs_of_parents ← select_pairs_of_individuals_to_be_crossed
                    (n_pairs_selected_parents) (1)
 // crossing over for each pair of parents
 children_parents ← create_two_children_for_each_selected_pair_of_parents
                    ( pairs_of_parents)
 perform_one_mutation_per_child(children_parents,prob_mutation)
 update_population(children_parents, pop,n)
 incr(i)
} // end while

// return at most k distinct individuals identified as the solutions with highest scores in the
last population
best_solutions ← identify_best_solutions(pop,k)

-------------------------------------------------------------------------------------------
Comments

(1) A basic version will allow each individual to be crossed over with another one.

A more advanced version will allow the user to specify either the percentage of individuals
in the population that are allowed to be crossed, or, equivalently, the number of pairs of
parents that are involved in a crossing over. In the two latter cases, the parents must be
selected according to probabilities proportional to the individuals' scores (roulette wheel).

A third version will consider a user-specified number s of r-way tournaments. A r-way
tournament selects at random r individuals from the population. The individual with the
best score is selected. Thus, s r-way tournaments provide s selected individuals. A user-
specified number of distinct pairs are then formed at random.

-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
```

```
PROCEDURE multi_objective()

  input
  n:                          number of individuals (i.e., solutions) in each population
  n_it:                       maximal number of iterations (i.e., successive populations)
  nc:                         number of individuals in the comparison set
  nt:                         number of tournaments for each iteration
  prob_mutation:              probability that a child be subject to a mutation
  freq_niching :              frequency to operate niching // niching is computationally
                              expensive


  output
  pareto_optimal_solutions:   a set of distinct solutions in the Pareto front (i.e. non-
                              dominated solutions)

pop ← initialize_population(n) ; i ← 1
comparison_set ← initialize_comparison_set(nc)

while (not termination_condition){

 pairs_of_parents ← select_pairs_of_individuals_by_tournament(n,nt,pop,comparison_set)

 // crossing over for each pair of parents
 children_parents ← create_two_children_for_each_selected_pair_of_parents
                    ( pairs_of_parents)
 perform_one_mutation_per_child(children_parents,prob_mutation)
 update_population(children_parents, pop,n)
 if (i mod freq_niching == 0){niching(pop)}
 incr(i)
} // end while

// return all non-dominated solutions in the last population
pareto_optimal_solutions ← identify_all_non_dominated_solutions(pop)

//*************************************************************************************
```

```
FUNCTION select_pairs_of_individuals_by_tournament(n,nt,pop, comparison_set):
pairs_of_parents

  // selection at random of n distinct pairs of individuals from current population
  select_at_random_n_distinct_pairs_of_individuals(pop,n)

  // implementation of the n tournaments in multi-objective framework
  selected_individuals ← empty_set
  for each pair pa of selected individuals{
    i1 ← first_individual(pa) ; i2 ← second_individual(pa)

    status1 ← is_dominated(i1,comparison_set)
    status2 ← is_dominated(i2,comparison_set)

    switch(status1){
      true : switch(status2){
               true : // no action
               false : // i2 is not dominated, i2 must be selected
                        add(selected_individuals, i2)
             }

      false : switch(status2){
               true : // i1 is not dominated, i1 must be selected
                       add(selected_individuals, i1)
               false : // no action
             }
    }  // end switch
  } // end for each

//*********************************************************************************************

FUNCTION is_dominated(individual,comparison_set): boolean
// postcondition :
// This function returns true if and only if individual is dominated by each element
// of comparion_set.

for each ind_comp in comparison_set{
 if (not is_dominated(individual,ind_comp)){
  return false
 } // end if
} // end for each
return true

//*********************************************************************************************
```

```
// Implementation in the case of maximization for the two objectives
FUNCTION is_dominated(ind1,ind2):boolean
// postcondition :
// This function returns true if and only if ind1 is dominated by ind2.

if (score1(ind2) > score1(ind1)){
 return (score2(ind2) ≥ score2(ind1))
}
// score1(ind2) <= score1(ind1)
if (score1(ind2) == score1(ind1)){
 return (score2(ind2) > score2(ind1))
}
// score1(ind2) < score1(ind1)
return false

//*****************************************************************************************

PROCEDURE niching(pop)

for each individual ind in pop{

 sum_share ← 0
 for each ind_other <> ind in pop{
  sum_share ← sum_share + share(ind,ind_other)
 } // end for each

// correct score (The score of ind mut be decreased if ind is close to a large number of
// individuals in the population. Thus, other areas around other local optima will have a
// chance to be identified.
 score(ind) ← score(ind) /  sum_share
} // end for each

//*****************************************************************************************

FUNCTION share(ind1,ind2) : real
// returns a normalized measure of similarity between ind1 and ind2

//*****************************************************************************************
```