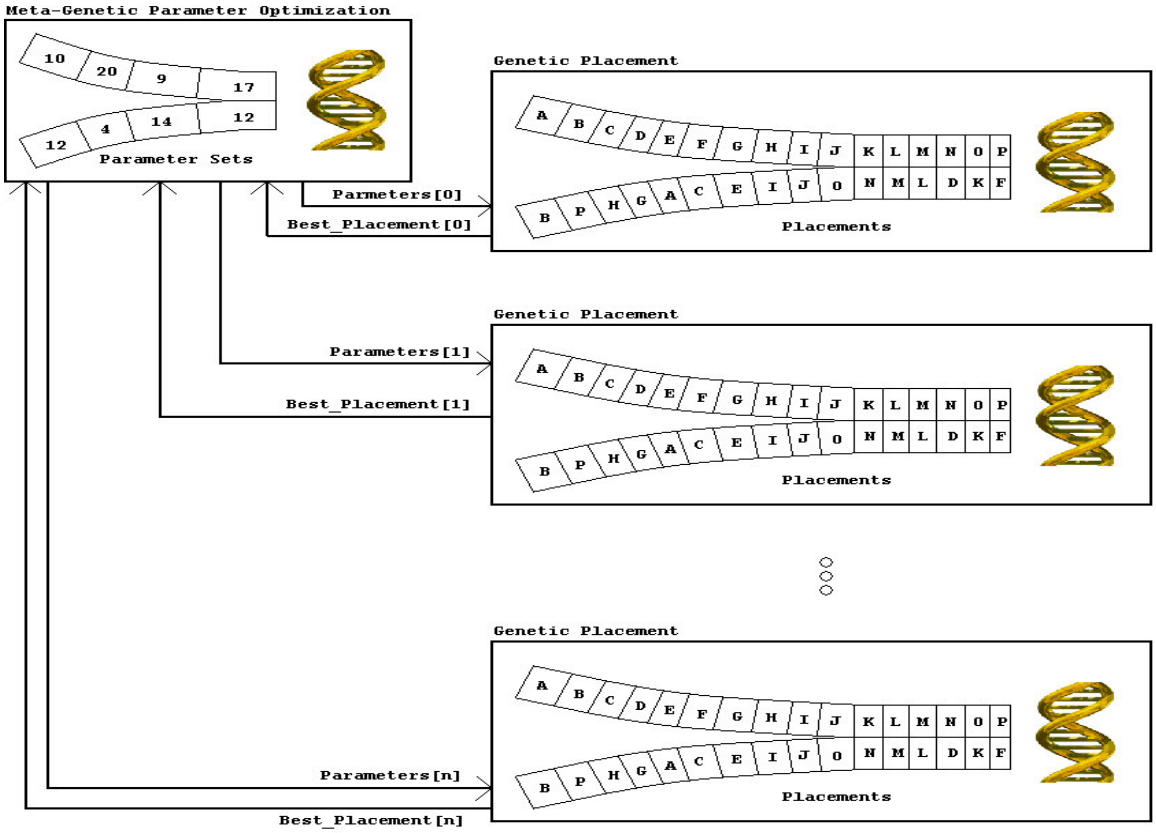
Report on the Implementation of GASP Using Meta-Genetic Parameter Optimization

Based on:
[1] K. Shahookar and P. Mazumder. A genetic approach to standard cell placement using meta-genetic parameter optimization. *IEEE Transactions on Computer Aided Design*, 9(5):500-511, May 1990.

I. Overview

The genetic technique for standard cell placement presented in [1] is a two-level, hierarchal random-search algorithm to find cell placements that have low routing cost. The lower level of the hierarchy consists of a genetic-based placement algorithm, taking a set of parameters and producing a locally optimal placement within the bounds of the parameters. The upper level is another genetic optimization process, but optimizes parameters given to the lower level, rather than placement configurations.



I.A. Genetic Algorithms

In general, genetic algorithms operate on strings of data which each represent a solution to the given problem. GAs maintain a dynamic population of solutions, usually "best" solutions, from which new solutions and future populations are formed. The method by which new solutions are generated resembles the combination process of biological DNA, hence the term "genetic."

To generate a new solution, two solutions are chosen from the population to serve as "parents" of the new solution. The parents then undergo a process known as "Crossover," in which some elements of each parent is given to the "offspring." The exact mechanism of crossover is implementation- and representation-specific, but the purpose of crossover is to allow each parent to contribute roughly half of its genetic data to its offspring.

After the crossover process has completed, the new offspring will "mutate" with some probability (usually a parameter to GAs). Mutation is intended to introduce new and untried genetic information into the population through random variation, just as biological mutation introduces variance in living populations.

Once some number of offspring have been generated through crossover and mutation, the combined population of parents and offspring are evaluated. The most fit members of this combined population are selected as potential parents of the next generation, and the algorithm continues iteratively. Other solutions are discarded.

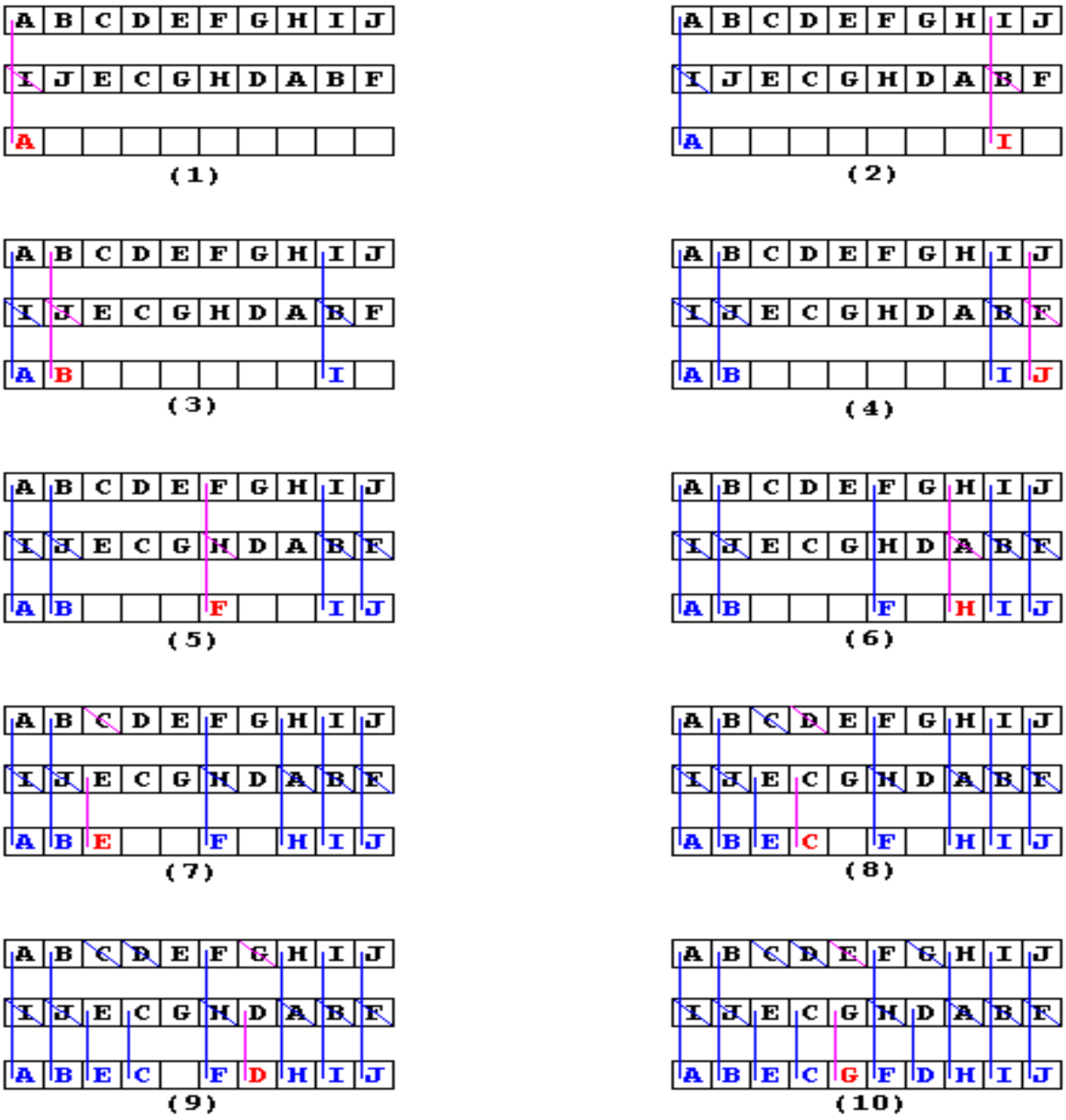
```
GeneticAlgorithm()
 Given: Np - Population Size, Ng - Number of Generations
        Rc - Crossover Rate,  Rm - Mutation Rate

 Generate Initial Population Randomly of size Np
 Evaluate Initial Population (give each member a fitness)

 // iterate over Ng iterations
 while(Generation < Ng )

   // spawn new children (selective breeding)
   while(nKids<Rc*Np)
     Spawn a new child through Crossover
     if(random < Rm)
       Mutate the child

     Evaluate the child (give child a fitness)
   endWhile
   Add the children to the population

   // survival of the fittest
   while(nKills<Rc*Np)
     Kill the least fit member of the population
   endwhile

   // there are now once again Np members of the population
 endwhile

 Return the most fit member of the population
endProcedure
```

I.B. Genetic Placement

Formulating the standard cell placement problem for use in a genetic algorithm depends entirely on the chosen representation of placements. To apply the genetic algorithm, the solution space must be representable as strings or sets of strings—Shahookar and Mazumder describe the solution space (all feasible placements) as an unordered set of triples (one triple per cell) that enumerate a cell's name (number), x-coordinate, and y-coordinate. A fourth quantity is used to quickly evaluate cell ordering (which otherwise would be determined from (x,y) coordinate pairs), but is not necessary for representation of the solution space. The population consists of many such sets of ordered triples, which are initially generated at random.

The use of unordered triples allows the authors to propose three different crossover operators: Order Crossover, PMX Crossover, and Cycle Crossover. Cycle crossover was empirically determined to be superior to the previous operators in producing children of greater genetic value. Only cycle crossover is included in this implementation.

In the offspring created by cycle crossover, each cell is in the same position as in one of its parents. If an arbitrary parent and position is chosen to begin with, this demands a cyclic progression. That is, if cells A and B occupy the same position in either parent, then selecting cell A from the first parent to be in its original position also means that the child must inherit B's position from that same parent. Inheriting B's position may further require that the child inherit cell C's position as well, and so on. This cycle ends when the non-chosen parent's cell already exists in the new solution. The role of the parents are then exchanged, and the crossover continues. To illustrate, consider crossing over the following two placements:



Suppose the top parent's first position is arbitrarily selected as a starting position. Cell A from the top parent is copied to the offspring (1). Because Cell I occupied position 1 in the bottom parent, the child must also inherit Cell I from the top parent, as in (2). Cell I occupies position 9 in the top parent, which is occupied by Cell B in the bottom parent. Hence, Cell B is also inherited from the top parent (3) at position 2. This cycle continues until (6), when the offspring inherits Cell H from the top parent at position 8. Note that Cell A occupies position 8 in the bottom parent. Cell A already appears in the offspring, so the role of the parents is reversed—the offspring will now inherit cell positions from the bottom parent. In (7) the offspring inherits cell E at position 3 from the bottom parent, thereby forcing the offspring to also inherit Cells C, D, and G from the bottom parent in (8), (9), and (10).

Mutation takes two forms in Shahookar and Mazumder's implementation of genetic placement. The first of these is cell swapping: two cells' positions in the placement are interchanged at random. This form of mutation changes the placement represented by the unordered triple. The second form is that of "inversion," in which some portion of the array containing the unordered triples is reversed. This reversal does *not* change the placement—only the representation thereof. Inversion tends to allow genetic data to be interchanged differently during crossover than if no inversion had occurred.

I.C. Meta-Genetic Optimization

Genetic placement constitutes only the lower level of the meta-genetic placement hierarchy; the upper level is itself another genetic process. In the higher level, the population consists of integer triples, representing crossover rate, mutation rate, and inversion rate variables that are passed to the lower level genetic placement algorithm during evaluation. The quality of the solution returned by genetic placement determines the fitness of the integer triples, and is used to drive the evolution of the parameters passed to the genetic placement algorithm.

Crossover of these triples is straightforward: the value of a given member of the triple for a new child is randomly chosen from the values belonging to the parents, with equal probability of selecting either parent. Mutation is also straightforward—a value between [-2,2] is added at random to one of the members of the triple—thereby introducing potentially new triples at each mutation.

As in the traditional genetic algorithm, individual fitness is used to determine which members of the combined population of parents and offspring will be used to produce the next generation of solutions.

II. Compilation

To compile: The command `make` in the presence of the implementation files will invoke the g++ compiler and linker as appropriate to generate the binary, `genetic`. The code has been compiled and tested under Windows 2000 / MS-DOS and Linux 2.4.26 environments, using the cl and gcc/g++ compilers, respectively.

III. Data Structures

In the genetic placement and meta-genetic portions of this implementations, the populations are represented as arrays of pointers. The members of the populations are also represented as arrays—arrays of fixed length (3) for the meta-genetic case, and sets of four arrays of variable length in the case of genetic placement.

This representation scheme is taken directly from the implementation described in [1]. A number of flag and storage variables were added to the representations to reduce redundant computation.

Arrays easily lend themselves to the genetic algorithm, due to their sequential, string-like access patterns. Arrays are ideal for representing strings of genetic information, and greatly simplify crossover and mutation operations.

IV. Pseudocode

The pseudocode of the algorithm has been partitioned into two sections: one for genetic placement, and one for the higher-level meta-genetic optimization process:

```
procedure MetaGenetic()
 Given: Np - Number of members in the population
        Ng - Number of generations
        Rc - Crossover rate
        Rm - Mutation rate

 1. population ← Np random triples
 2. Call Genetic() for each population member,
      a. Store the fitness of the placement returned as the
         population member's fitness
 3. nGenerations ← 0
 4. nKids ← 0
 5. Select two random parents, A and B
 6. Perform crossover on A and B, produce child C[nKids]
 7. If( random < Rm ), Mutate C[nKids]
 8. nKids ← nKids + 1
 9. if(nKids < Np * (1 + Rc)) GOTO 5
 10.Call Genetic() for each child in C,
      a. Store the fitness of the placement retuned as the
         child's fitness
 11.Add children C to the population of parents
 12.nRemovals ← 0
 13.Remove the least fit member of the combined population
    from the population
 14.nRemovals ← nRemovals + 1
 15.if( nRemovals < nKids) GOTO 13
 16.nGenerations ← nGenerations + 1
 17.if( nGenerations < Ng ) GOTO 4
endprocedure MetaGenetic()
```

```
procedure Genetic()
 Given: Np - Number of members in the population rate
        Ng - Number of generations
        Rc - Crossover rate
        Ri - Inversion rate
        Rm - Mutation rate
 1. population ← Np random placements
 2. Determine fitness of each population member (inverse
    wire length)
 3. nGenerations ← 0
 4. Perform inversion on Rc*Np members of the population at
    random
 5. nKids ← 0
 6. Select two parents A and B randomly with probabilities
    proportional to their fitnesses
 7. Perform crossover on A and B to produce child C[nKids]
      a. If( C[nKids] is a clone of A or B ), GOTO 6
 8. if( random < Rm ) Mutate child C[nKids]
 9. nKids ← nKids + 1
 10.if( nKids < Rc * Np ) GOTO 6
 11.Determine the fitness of all children in C (inverse
    wire length)
 12.Add children C to the population of parents
 13.nRemovals ← 0
 14.Remove the least fit member of the combined population
    from the population
 15.nRemovals ← nRemovals + 1
 16.if( nRemovals < nKids ) GOTO 14
 17.nGenerations ← nGenerations + 1
 18.if( nGenerations < Ng ) GOTO 5
endprocedure Genetic()
```

As is evident from the pseudocode, the key difference between the genetic placement algorithm and the meta-genetic optimization algorithm is that in the genetic placement algorithm the population consists of placements. In the meta-genetic optimization algorithm the population consists of configurations. The structure of the algorithms are otherwise identical.

V. Structure and Organization

The call stack of the implementation follows graphically below. To narrate, the main function (in main_genetic.cpp) does some initial input parsing and checking, then begins keeping time. Main() then calls metagenetic() (in metagenetic.cpp), which constitutes the top-level optimization described above. Metagenetic() will make several calls to Genetic() in the course of its execution. Genetic() (found in genetic.cpp) performs the

actual genetic placement, employing crossover and mutation functions Crossover(), Randomize(), Mutate(), and Invert() (all of which are found in placement.cpp).

```
int main()
     -Input parsing
     -Start time
     -Call Metagenetic()
          -Do algorithm as described above
          -Many calls to Genetic()
               - Do algorithm as described above
               - Calls to:
                    - Crossover()
                    - Randomize()
                    - Mutate()
                    - Invert()
               - Return best placement
          -Keep track of best placement found
          -Print the best placement
          -Return
     -Stop time
     -Print running time
     -Exit
```

VI. Execution

The implementation is command-line driven. The binary executable `genetic` allows the user to specify parameters to the algorithm or to use built-in defaults. A description of command-line options and syntax follows.

Syntax:
`genetic -i <input filename> [<switch> <switch argument>] …`

Switch	Argument	Bounds	Default Value
`-i`	Input file name	<string>	NONE
`-o`	Output file name	<string>	(terminal)
`-np`	Meta-Genetic population size	[1,∞)	20
`-ng`	Meta-Genetic generations	[1,∞)	10
`-npg`	Genetic population size	[1,∞)	10
`-ngg`	Genetic generations	[1,∞)	10
`-rc`	Meta-Genetic crossover rate	[0.0, 1.0]	1.0
`-rm`	Meta-Genetic mutation rate	[0.0, 1.0]	0.2
`-s`	Random seed	[0, ∞)	Time()
`-v`	Verbosity	[0,10]	0

The last two switches allow the user to customize the para-algorithmic behavior of execution: specifying a random seed ensures deterministic execution, and verbosity specifies what information is printed to the specified output file.

Verbosity	Printed Information
0	Meta-Genetic input parameters, final results, execution time
1	Meta-Genetic and Genetic error conditions
2	Meta-Genetic best solution updates
3	Meta-Genetic initial solution fitnesses
4	Meta-Genetic generated fitnesses and announcement of discarded configurations
5	Detailed information of each child in Meta-Genetic
6	Genetic() best solution updates
7	Genetic() fitness of each population member
8	Genetic() detailed fitnesses
9	Genetic() placements of each population member
10	Genetic() parameters passed from MetaGenetic()

Note: Large specified verbosities cause increasingly larger output files and increasingly longer runtimes.

Execution time varies greatly with the specified parameters. In general, larger NG, NP, NPG, and NGG values tend to yield increased runtimes proportional to the square of the increase. For large parameters, the most important parameter for determining runtime is verbosity, as large verbosities will result in many system calls and degrade performance. For best performance, leave verbosity at its default value of 0.

VI.A. Suggested Executions

To observe the meta-genetic optimization process, the following execution is suggested:

```
genetic –i gen_10_1.txt –np 3 –ng 5 –v 4
```

To observe the process of genetic placement, the following execution is suggested:

```
genetic –i gen_10_1.txt –np 1 –ng 1 –npg 10 –ngg 5 –v 7
```

To observe larger-scale interactions between the meta-genetic optimization and placement algorithms, the following execution is suggested:

```
genetic –i gen_50_1.txt –v 4
```

VII. Improvements

An effective method to improve performance of the genetic algorithm is to ensure that the greatest amount of variation is present at any given time in execution. The presence of diversity tends to cause maximal exploration of the available solution space in a

relatively short time. To that end, I propose the following modifications to the algorithms presented above:

1) Initial populations should not be generated randomly. Instead, only the first and second population member should be generated at random. Subsequent members of the initial population should be generated through *anti-crossover*. The purpose of anti-crossover is to generate a solution that is genetically very different from its two parent solutions. Anti-crossover could use mechanisms similar to but opposite of traditional crossover. It may even be worthwhile to generate the initial population iteratively and genetically, employing anti-crossover in place of crossover and eliminating population members that are not sufficiently diverse.
2) The genetic algorithm could be combined with the concept of simulated annealing. While the temperature of the annealing process is "high," the genetic algorithm is likely to allow inferior but more diverse solutions to remain in the breeding population, and may eliminate some superior solutions to maintain this diversity. As the temperature "cools," the genetic algorithm would be more and more likely to eliminate inferior solutions. This would allow for a very rapid exploration of the solution space early in the algorithm, and more detailed examination in the final cycles of the algorithm.
3) Instead of simply disallowing clones of parents as in the Genetic() procedure above, any generated clones should be automatically replaced with completely new, random solutions. The automatic replacement of clones with random solutions should cause the genetic algorithm to "self-pollinate" with new genetic material as it is needed. The same could be applied to the MetaGenetic() process, though it is less likely to "breed itself into a corner."