A biased random-key genetic algorithm for the Steiner triple covering problem

Talk given at the 9th Metaheuristics International Conference
Udine, Italy ✤ July 28, 2011

Mauricio G. C. Resende
AT&T Labs Research
Florham Park, New Jersey

mgcr@research.att.com

Joint work with R. Toso, J.F. Gonçalves, and R.M.A. Silva

This is a MIC 2011 category S3 talk (recently published papers)

M.G.C. Resende, R.F. Toso, J.F. Gonçalves, and R.M.A. Silva, "A biased random-key genetic algorithm for the Steiner triple covering problem," Optimization Letters, published online 06 February 2011.

Summary

- Steiner triple covering problem
- Biased random-key genetic algorithms (BRKGA)
- BRKGA for Steiner triple covering problem
- Implementation issues
- Experimental results
- Concluding remarks

Steiner triple covering problem

Kirkman school girl problem [Kirkman, 1850]

Fifteen young ladies in a school walk out three abreast for seven days in succession:

It is required to arrange them daily, so that no two shall walk twice abreast.

Kirkman school girl problem [Kirkman, 1850]

If girls are numbered 01, 02, ..., 15, a solution is:

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
01, 06, 11	01, 02, 05	02, 03, 06	05, 06, 09	03, 05, 11	05, 07, 13	04, 11, 13
02, 07, 12	03, 04, 07	04, 05, 08	07, 08, 11	04, 06, 12	06,08, 14	05, 12, 14
03, 08, 13	08, 09, 12	09, 10, 13	01, 12, 13	07, 09, 15	02, 09, 11	02, 08, 15
04, 09, 14	10, 11, 14	11, 12, 15	03, 14, 15	01, 08, 10	03, 10, 12	01, 03, 09
05, 10, 15	06, 13, 15	01, 07, 14	02, 04, 10	02, 13, 14	01, 04, 15	06, 07, 10

Ball, Rouse, and Coxeter (1974)

Steiner triple system

A Steiner triple system on a set X of n elements is a collection B of 3-sets (triples) such that, for any two elements x and y in X, the pair {x, y} appears in exactly one triple in B.

Steiner triple system

A Steiner triple system on a set X of n elements is a collection B of 3-sets (triples) such that, for any two elements x and y in X, the pair {x, y} appears in exactly one triple in B.

First studied by Kirkman in 1847. Then by Steiner in 1853 and hence the name.

Steiner triple system

A Steiner triple system on a set X of n elements is a collection B of 3-sets (triples) such that, for any two elements x and y in X, the pair {x, y} appears in exactly one triple in B.

The school girl problem has the additional constraint that the collection of $|B| = 7 \times 5 = 35$ triples be divided into seven sets of five triples, one for each day, such that each girl appears exactly once in the set of five triples for that day.

Steiner triple system

A Steiner triple system on a set X of n elements is a collection B of 3-sets (triples) such that, for any two elements x and y in X, the pair {x, y} appears in exactly one triple in B.

A Steiner triple system exists for a set X if and only if either $|X| = 6k+1$ or $|X| = 6k+3$ for some $k > 0$ [Kirkman, 1847]

Steiner triple system

A Steiner triple system on a set X of n elements is a collection B of 3-sets (triples) such that, for any two elements x and y in X, the pair {x, y} appears in exactly one triple in B.

One non-isomorphic Steiner triple system exists for $|X| = 7$ and 9. This number grows quickly after that. For $|X| = 19$, there are over 10^{10} non-isomorphic Steiner triple systems.

Steiner triple system

A Steiner triple system on a set X of n elements is a collection B of 3-sets (triples) such that, for any two elements x and y in X, the pair {x, y} appears in exactly one triple in B.

A Steiner triple system can be represented by a binary matrix A with one column for each element in X and a row for each triple in B. In this matrix A(i,j) = 1 if and only if element j is in triple i.

Each row i of A has exactly 3 entries with A(i,j) = 1.

1-width of a binary matrix

The 1-width of a binary matrix A is the minimum number of columns that can be chosen from A such that every row has at least one "1" in the selected columns.

The 1-width of a binary matrix A is the solution of the set covering problem: $\min \sum_j x_j$ subject to $Ax \geq 1_m$, $x_j \in \{0, 1\}$

Recursive procedure to generate Steiner triple systems

Let A_3 be the 1×3 matrix of all ones. A recursive procedure described by Hall (1967) can generate Steiner triple systems for which $n = 3^k$ or $n = 15 \times 3^{k-1}$, for $k = 1, 2, \ldots$

Recursive procedure to generate Steiner triple systems

Let A_3 be the 1×3 matrix of all ones. A recursive procedure described by Hall (1967) can generate Steiner triple systems for which $n = 3^k$ or $n = 15 \times 3^{k-1}$, for $k = 1, 2, \ldots$

Starting from A_3, the procedure can generate A_9, A_{27}, A_{81}, A_{243}, A_{729}, ...

Starting from A_{15} [Fulkerson et al., 1974], the procedure can generate A_{45}, A_{135}, A_{405}, ...

Solving Steiner triple covering

Fulkerson, Nemhauser, and Trotter (1974) were first to point out that the Steiner triple covering problem was a computationally challenging set covering problem.

Solving Steiner triple covering

Fulkerson, Nemhauser, and Trotter (1974) were first to point out that the Steiner triple covering problem was a computationally challenging set covering problem.

They solved stn9 (A_9), stn15 (A_{15}), and stn27 (A_{27}) to optimality, but not stn45 (A_{45}), which was solved in 1979 by Ratliff.

Mannino and Sassano (1995) solved stn81 and recently Ostrowski et al. (2009; 2010) solved stn135 in 126 days of CPU and stn243 in 51 hours. Independently, Ostergard and Vaskelainen (2010) also solved stn135.

Heuristics for Steiner triple covering (stn81 and stn135)

- Feo and R. (1989) proposed a GRASP, finding a cover of size 61 for stn81, later shown to be optimal by Mannino and Sassano (1995).

Heuristics for Steiner triple covering (stn81 and stn135)

- Feo and R. (1989) proposed a GRASP, finding a cover of size 61 for stn81, later shown to be optimal by Mannino and Sassano (1995).
- Karmarkar, Ramakrishnan, and R. (1991) found a cover of size 105 for stn135 with an interior point algorithm. In the same paper, they used a GRASP to find a better cover of size 104. Mannino and Sassano (1995) also found a cover of this size.

Heuristics for Steiner triple covering (stn81 and stn135)

- Feo and R. (1989) proposed a GRASP, finding a cover of size 61 for stn81, later shown to be optimal by Mannino and Sassano (1995).
- Karmarkar, Ramakrishnan, and R. (1991) found a cover of size 105 for stn135 with an interior point algorithm. In the same paper, they used a GRASP to find a better cover of size 104. Mannino and Sassano (1995) also found a cover of this size.
- Odijk and van Maaren (1998) found a cover of size 103, which was shown to be optimal by Ostrowski et al. and Ostergard and Vaskelainen in 2010.

Heuristics for Steiner triple covering (stn243)

- The GRASP in Feo and R. (1989) as well as the interior point method in Karmarkar, Ramakrishnan, and R. (1991) produced covers of size 204 for stn243.

Heuristics for Steiner triple covering (stn243)

- The GRASP in Feo and R. (1989) as well as the interior point method in Karmarkar, Ramakrishnan, and R. (1991) produced covers of size 204 for stn243.
- Karmarkar, Ramakrishnan, and R. (1991) used the GRASP of Feo and R. (1989) to improve the best known cover to 203.

Heuristics for Steiner triple covering (stn243)

- The GRASP in Feo and R. (1989) as well as the interior point method in Karmarkar, Ramakrishnan, and R. (1991) produced covers of size 204 for stn243.
- Karmarkar, Ramakrishnan, and R. (1991) used the GRASP of Feo and R. (1989) to improve the best known cover to 203.
- Mannino and Sassano (1995) improved it further to 202.

Heuristics for Steiner triple covering (stn243)

- The GRASP in Feo and R. (1989) as well as the interior point method in Karmarkar, Ramakrishnan, and R. (1991) produced covers of size 204 for stn243.
- Karmarkar, Ramakrishnan, and R. (1991) used the GRASP of Feo and R. (1989) to improve the best known cover to 203.
- Mannino and Sassano (1995) improved it further to 202.
- Odijk and van Maaren (1998) found a cover of size 198, which was shown to be optimal by Ostrowski et al. (2009; 2010).

Heuristics for Steiner triple covering (stn405 and stn729)

- No results have been previously presented for stn405.

Heuristics for Steiner triple covering (stn405 and stn729)

- No results have been previously presented for stn405.
- Ostrowski et al. (2010) report that the best solution found by CPLEX 9 on stn729 after two weeks of CPU time was 653.

Heuristics for Steiner triple covering (stn405 and stn729)

- No results have been previously presented for stn405.
- Ostrowski et al. (2010) report that the best solution found by CPLEX 9 on stn729 after two weeks of CPU time was 653.
- Using their enumerate-and-fix heuristic, they were able to find a better cover of size 619.

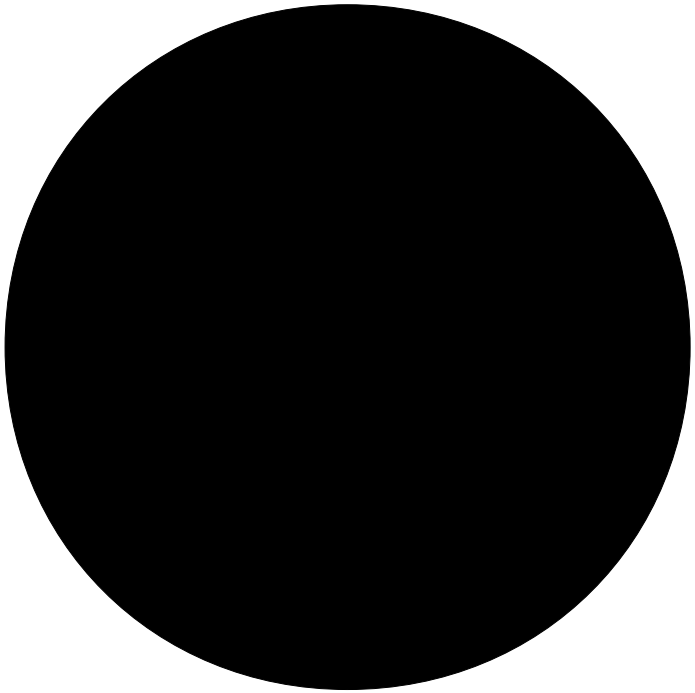
Best known solutions to date

instance	n	m	BKS	opt?	reference
stn9	9	12	5	yes	Fulkerson et al. (1974)
stn15	15	35	9	yes	Fulkerson et al. (1974)
stn27	27	117	18	yes	Fulkerson et al. (1974)
stn45	45	330	30	yes	Ratliff (1979)
stn81	81	1080	61	yes	Mannino and Sassano (1995)
stn135	135	3015	103	yes	Ostrowski et al. (2009; 2010) and Ostergard and Vaskelainen (2010)
stn243	243	9801	198	yes	Ostrowski et al. (2009; 2010)
stn405	405	27270	335	?	This paper.
stn729	729	88452	617	?	This paper.

Biased random-key genetic algorithms

Genetic algorithms

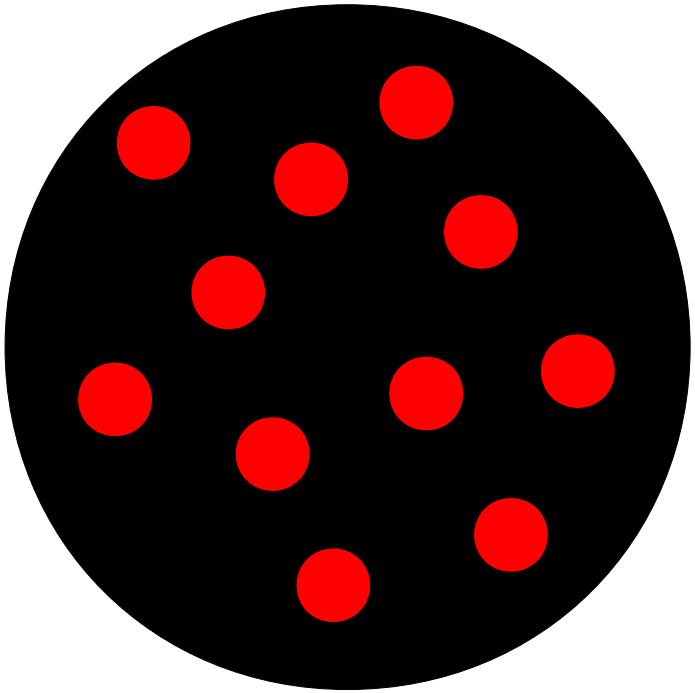
Holland (1975)



Adaptive methods that are used to solve search and optimization problems.

Individual: solution 

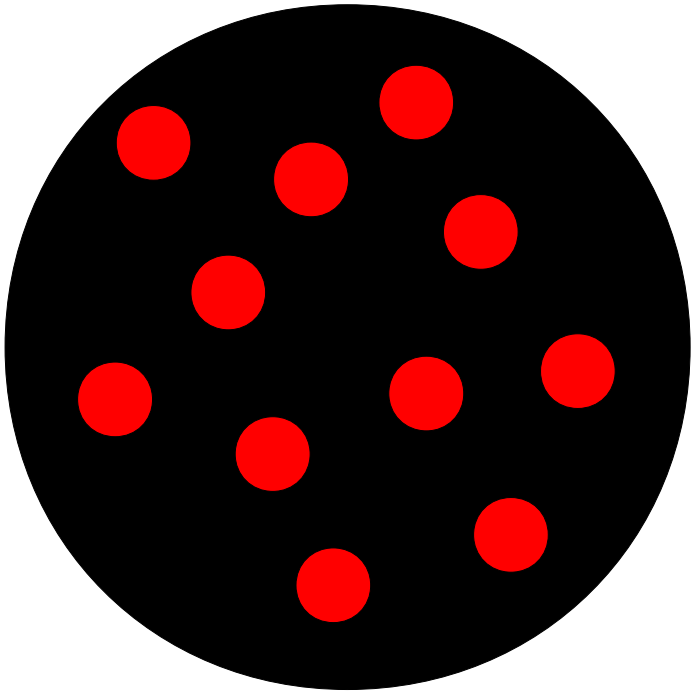
Genetic algorithms



Individual: solution

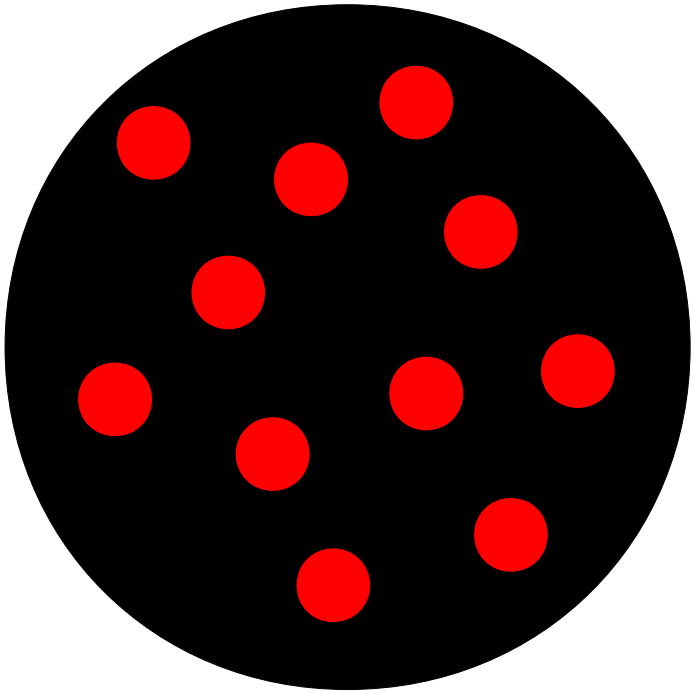
Population: set of fixed number of individuals

Genetic algorithms



Genetic algorithms evolve population applying the principle of survival of the fittest.

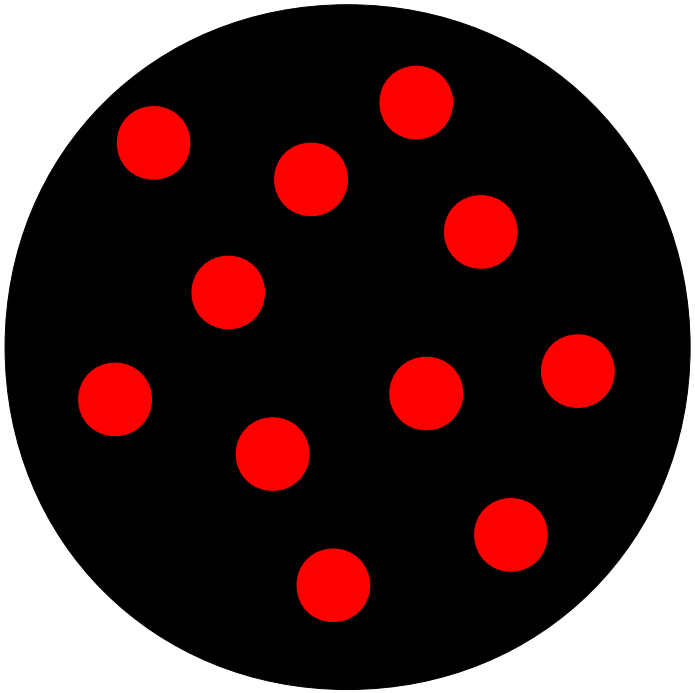
Genetic algorithms



Genetic algorithms evolve population applying the principle of survival of the fittest.

A series of generations are produced by the algorithm. The most fit individual of last generation is the solution.

Genetic algorithms

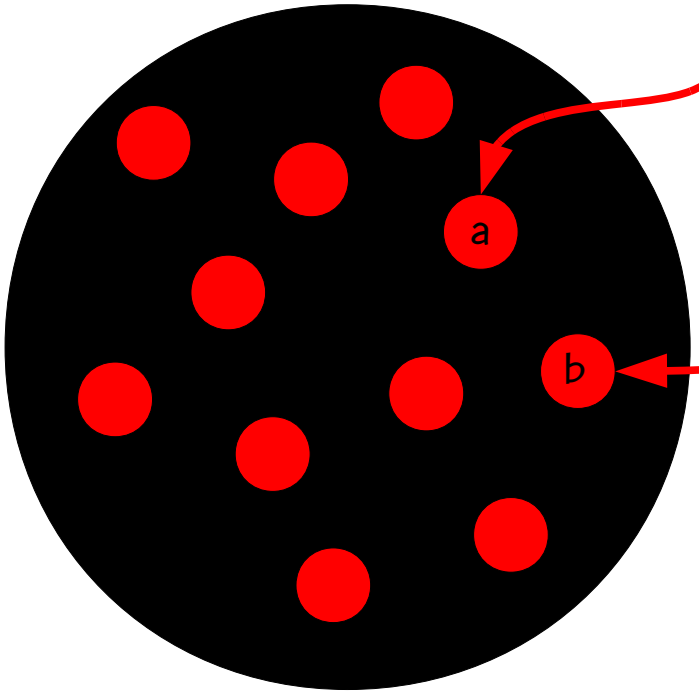


Genetic algorithms evolve population applying the principle of survival of the fittest.

A series of generations are produced by the algorithm. The most fit individual of last generation is the solution.

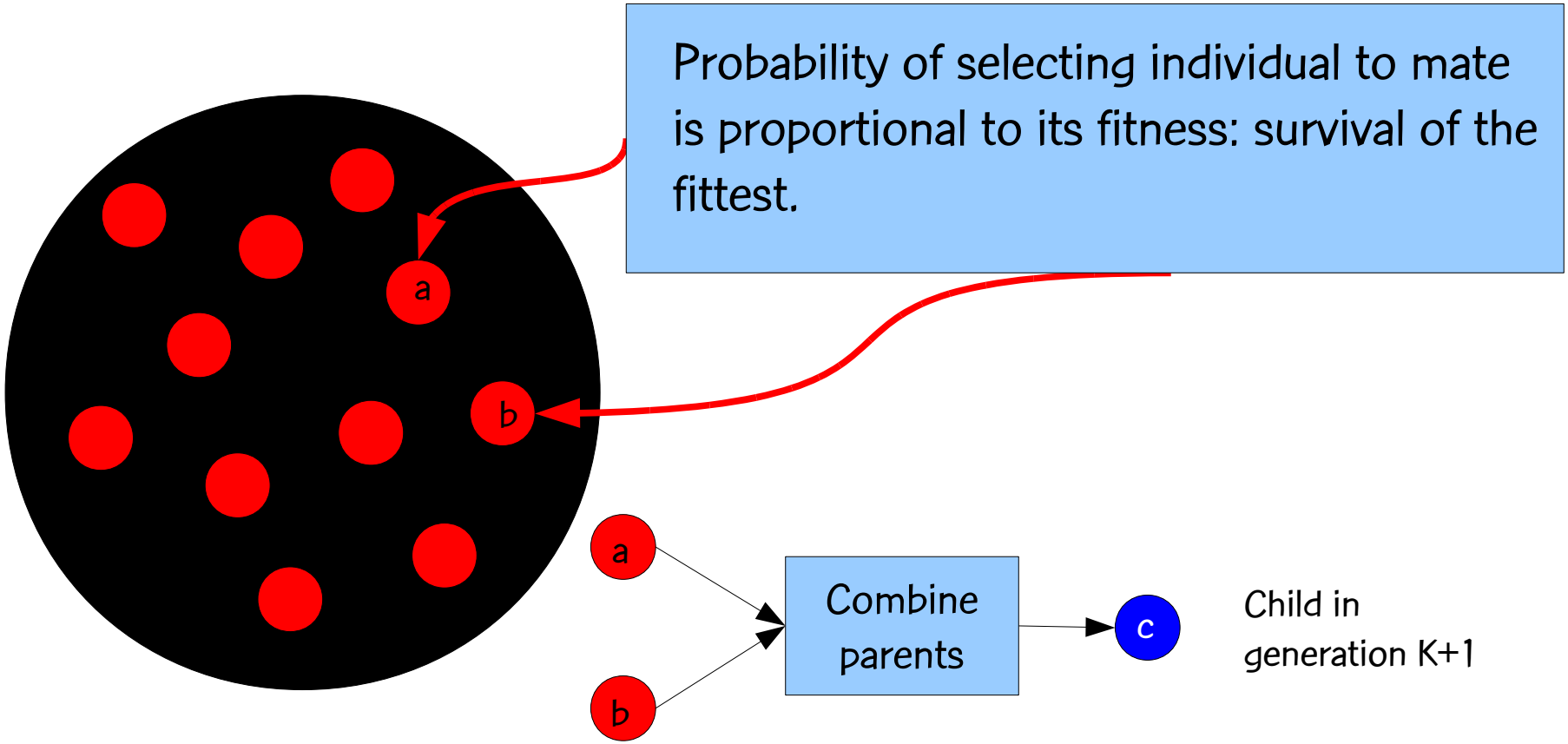
Individuals from one generation are combined to produce offspring that make up next generation.

Genetic algorithms

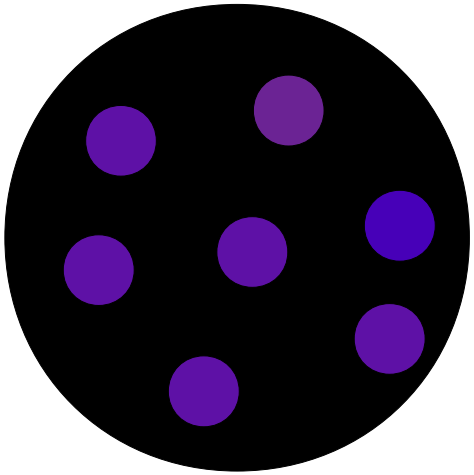


Probability of selecting individual to mate is proportional to its fitness: survival of the fittest.

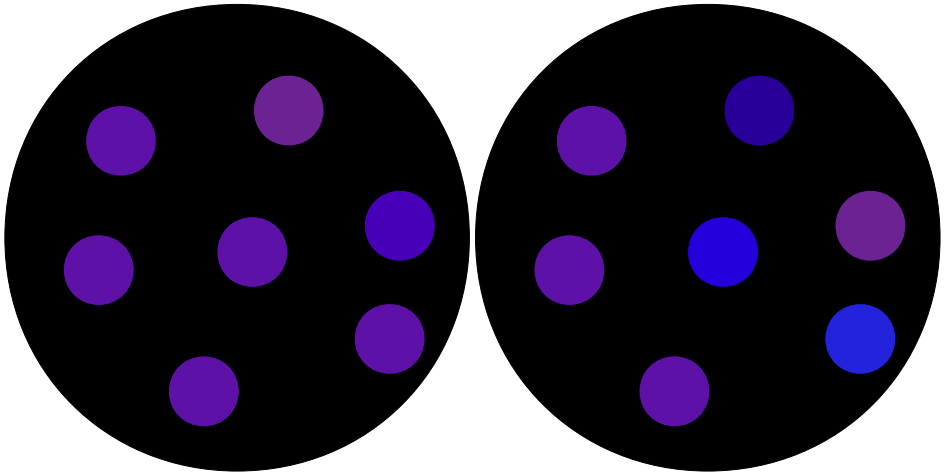
Genetic algorithms



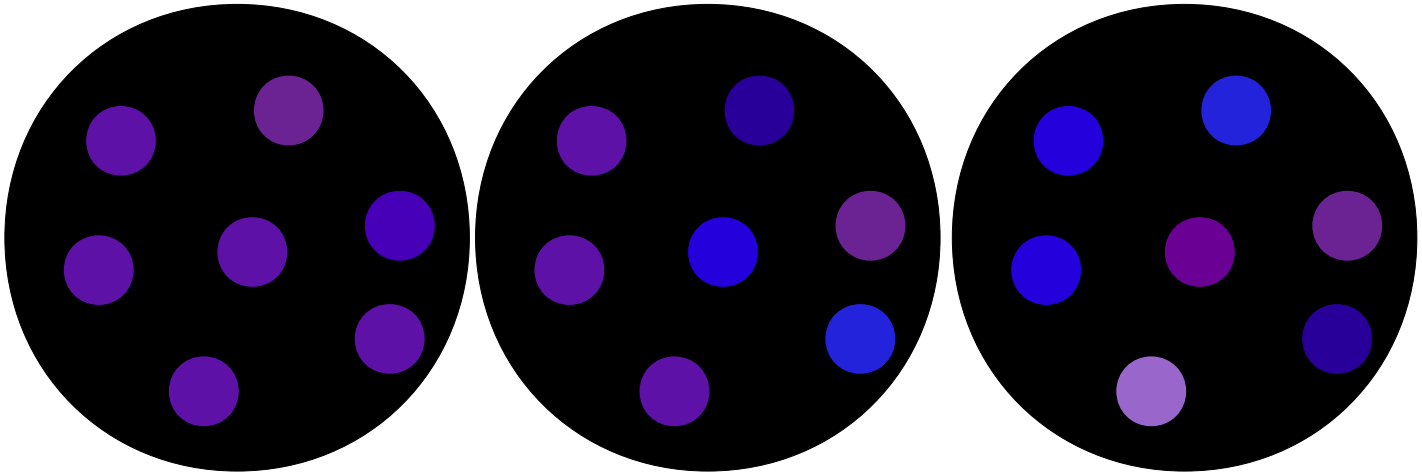
Evolution of solutions



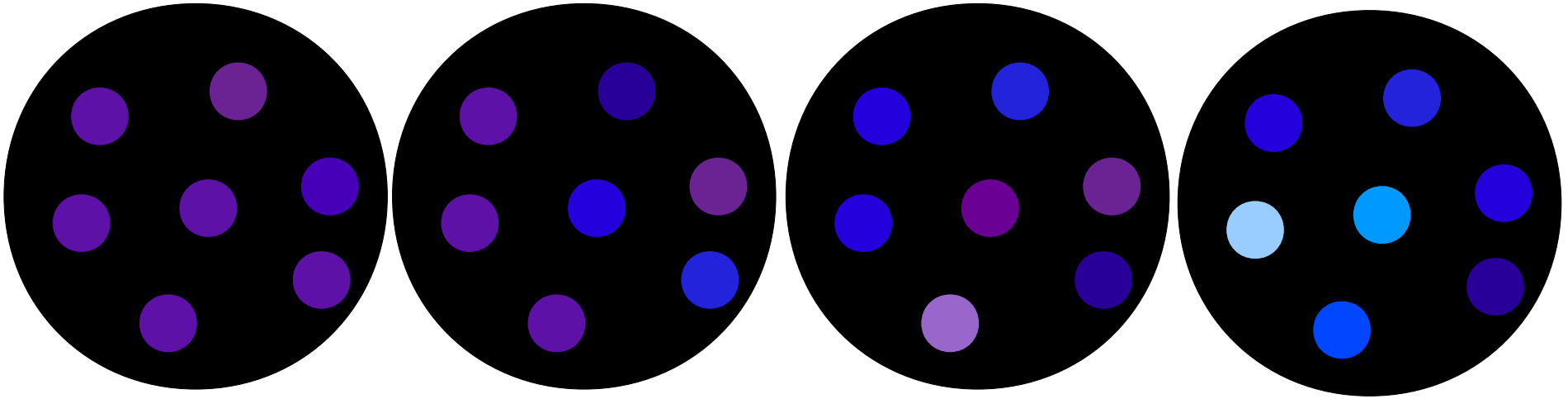
Evolution of solutions



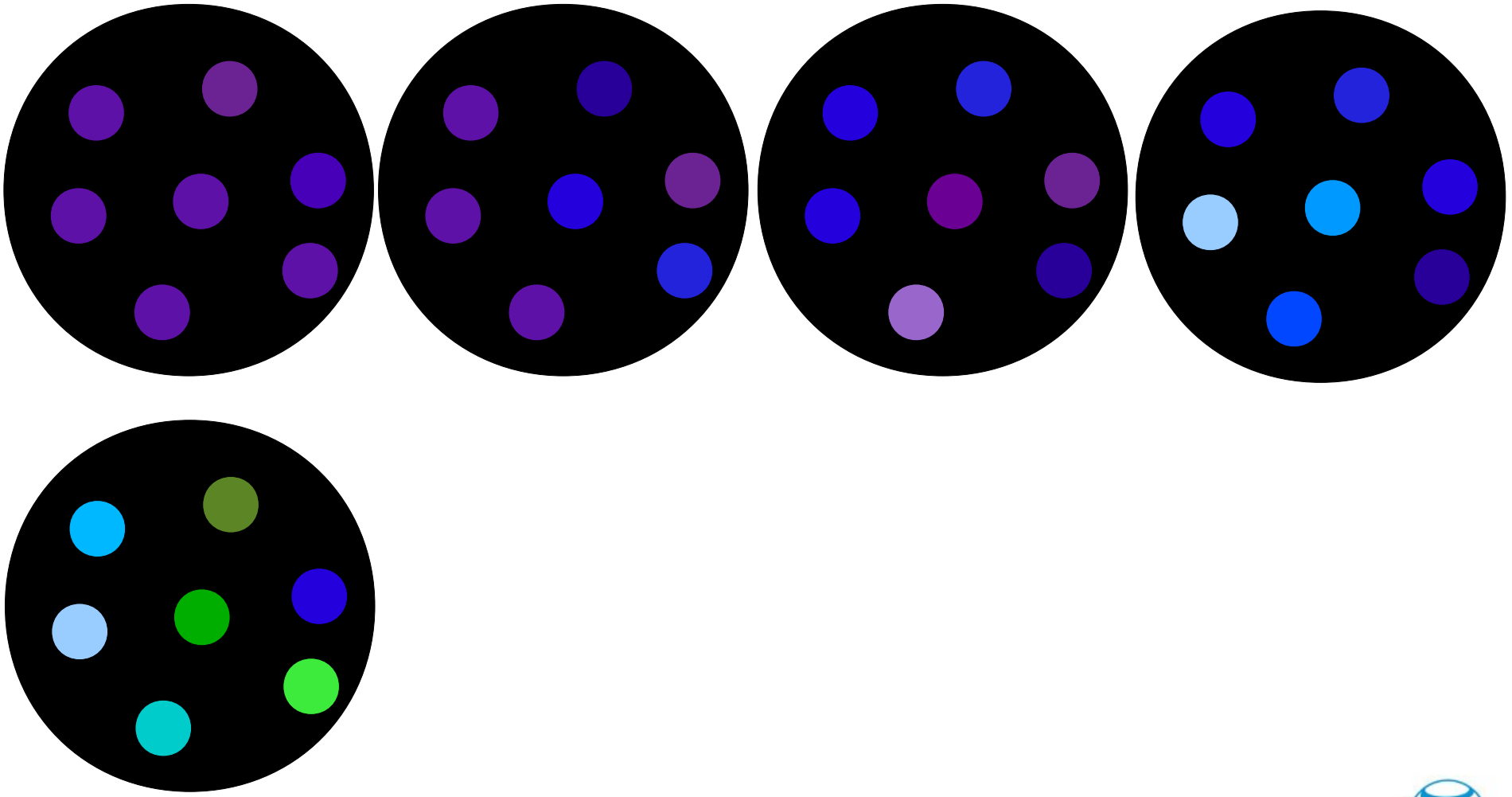
Evolution of solutions



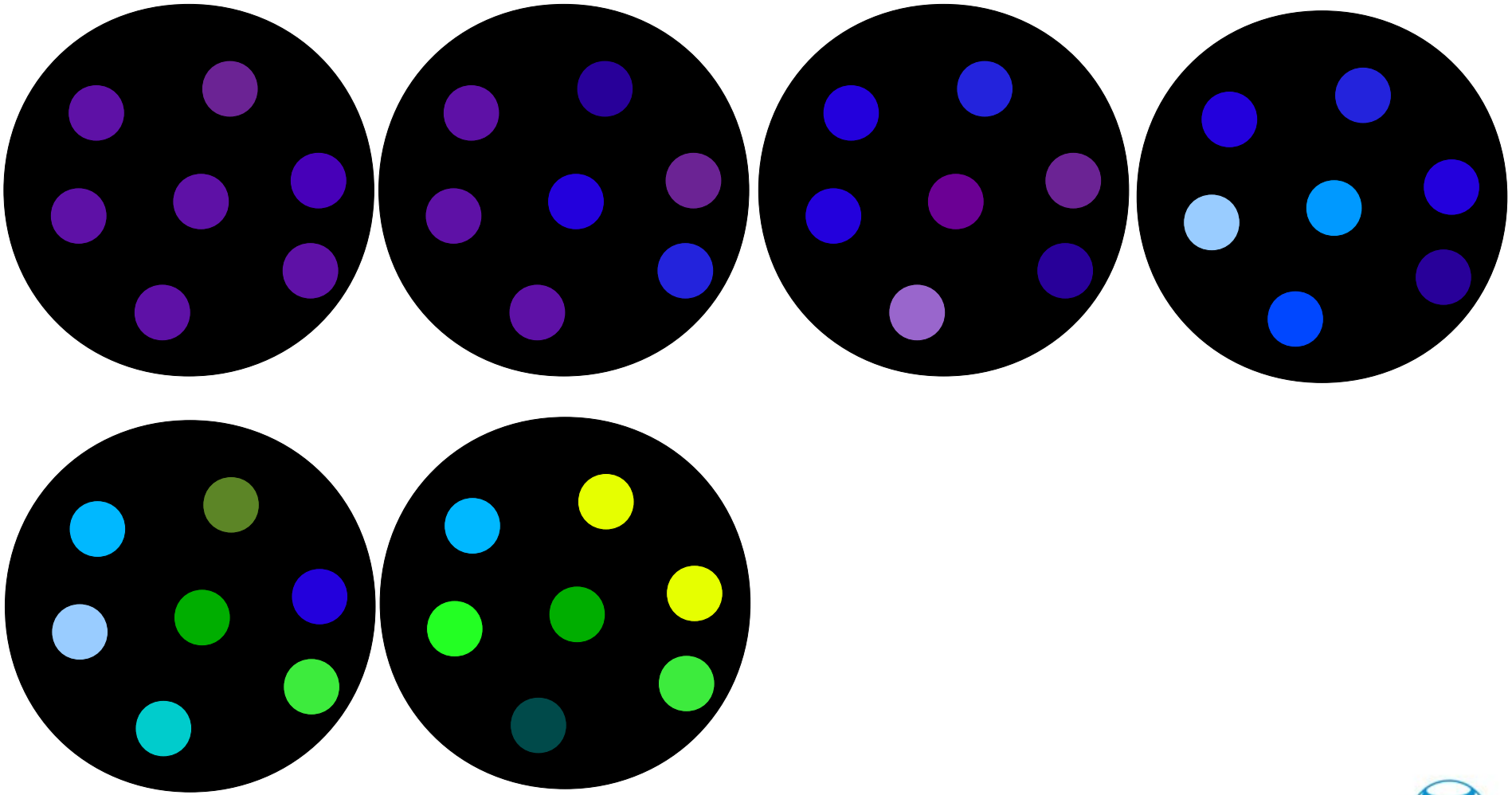
Evolution of solutions



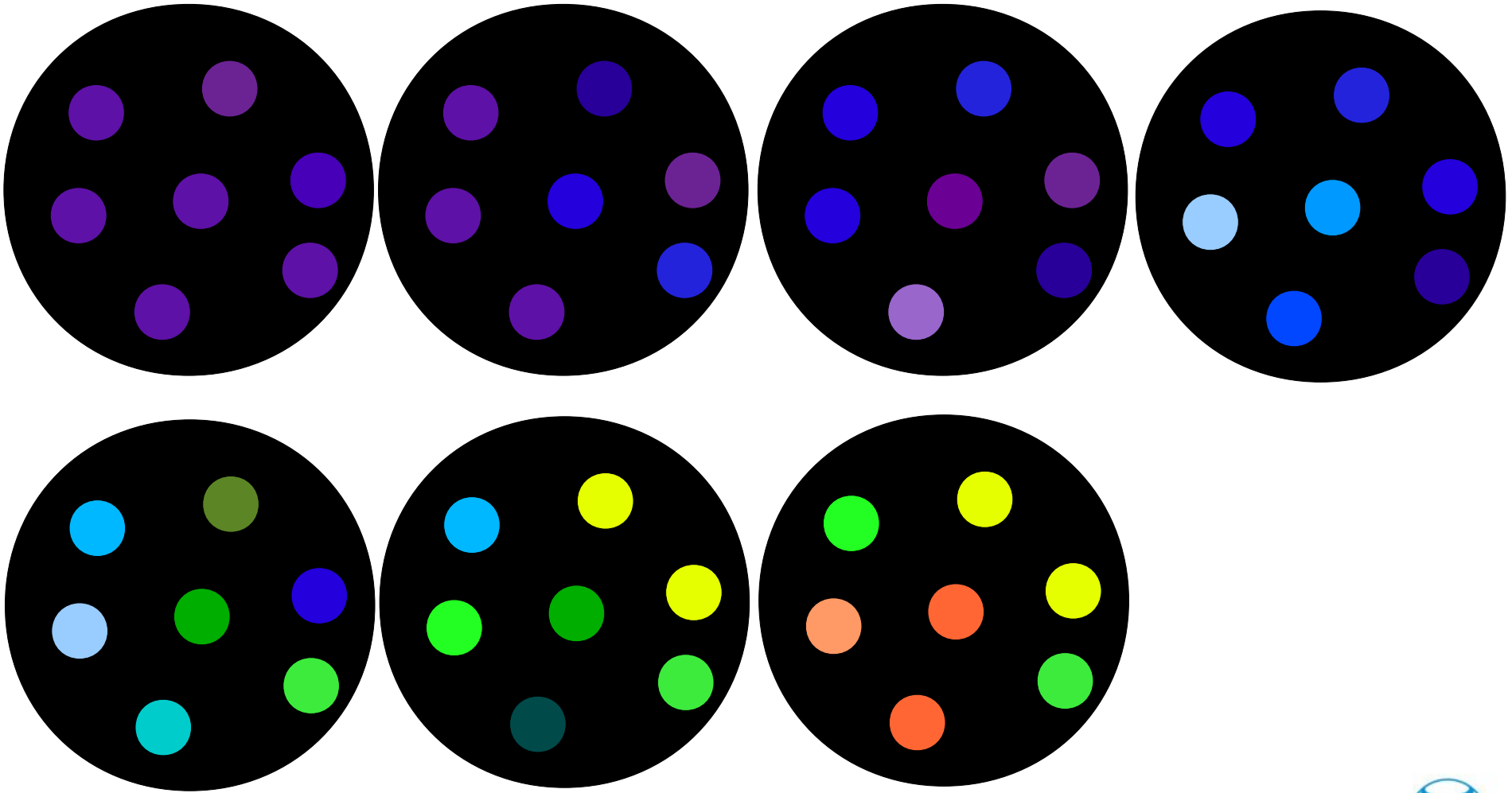
Evolution of solutions



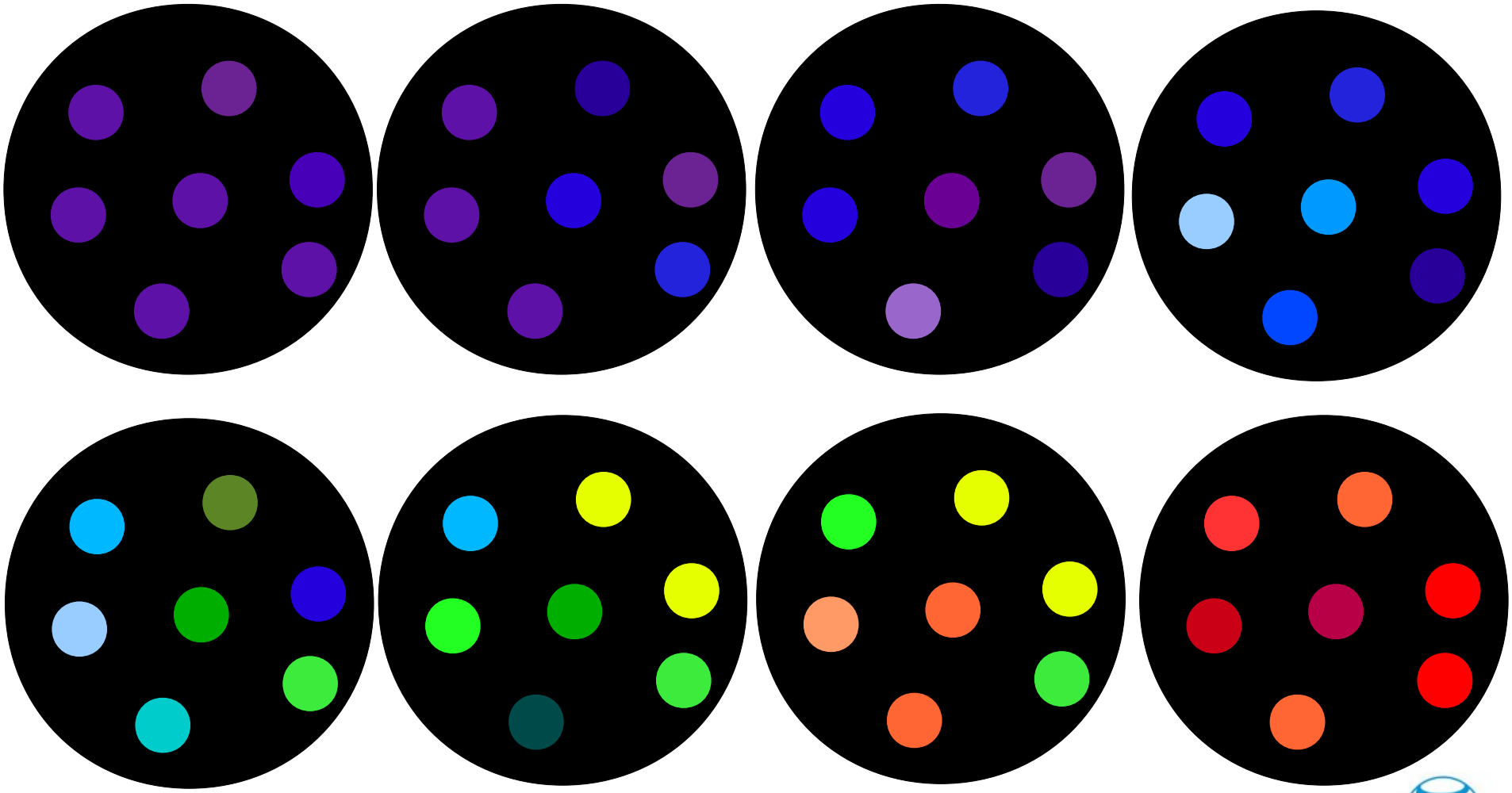
Evolution of solutions



Evolution of solutions



Evolution of solutions



Genetic algorithms with random keys

GAs and random keys

- Introduced by Bean (1994) for sequencing problems.

GAs and random keys

- Introduced by Bean (1994) for sequencing problems.
- Individuals are strings of real-valued numbers (random keys) in the interval [0,1].

S = (0.25, 0.19, 0.67, 0.05, 0.89)
s(1) s(2) s(3) s(4) s(5)

GAs and random keys

- Introduced by Bean (1994) for sequencing problems.
- Individuals are strings of real-valued numbers (random keys) in the interval [0,1].
- Sorting random keys results in a sequencing order.

S = (0.25, 0.19, 0.67, 0.05, 0.89)
s(1) s(2) s(3) s(4) s(5)

S' = (0.05, 0.19, 0.25, 0.67, 0.89)
s(4) s(2) s(1) s(3) s(5)

Sequence: 4 – 2 – 1 – 3 – 5

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)
c = ( )

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)
c = (0.25)

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)
c = (0.25, 0.90)

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)
c = (0.25, 0.90, 0.76)

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)
c = (0.25, 0.90, 0.76, 0.05)

GAs and random keys

- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)
c = (0.25, 0.90, 0.76, 0.05, 0.89)

GAs and random keys

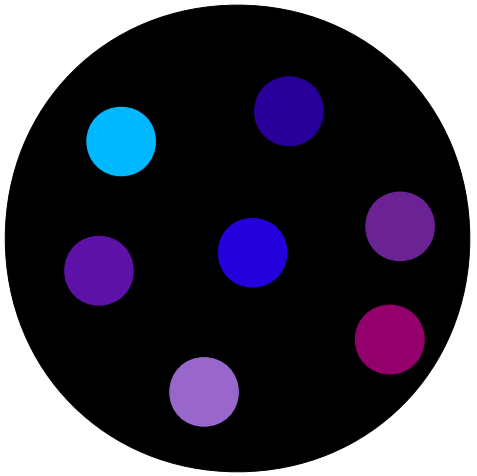
- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele to the child.

a = (0.25, 0.19, 0.67, 0.05, 0.89)
b = (0.63, 0.90, 0.76, 0.93, 0.08)
c = (0.25, 0.90, 0.76, 0.05, 0.89)

If every random-key array corresponds to a feasible solution: Mating always produces feasible offspring.

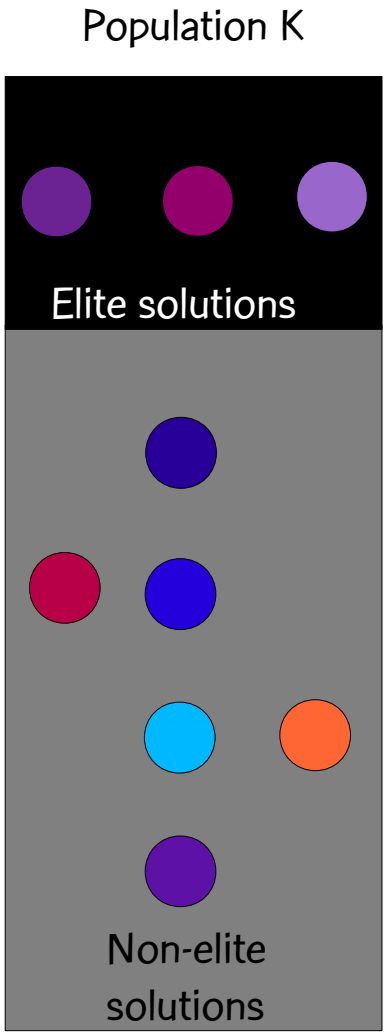
GAs and random keys

Initial population is made up of P chromosomes, each with N genes, each having a value (allele) generated uniformly at random in the interval [0,1].



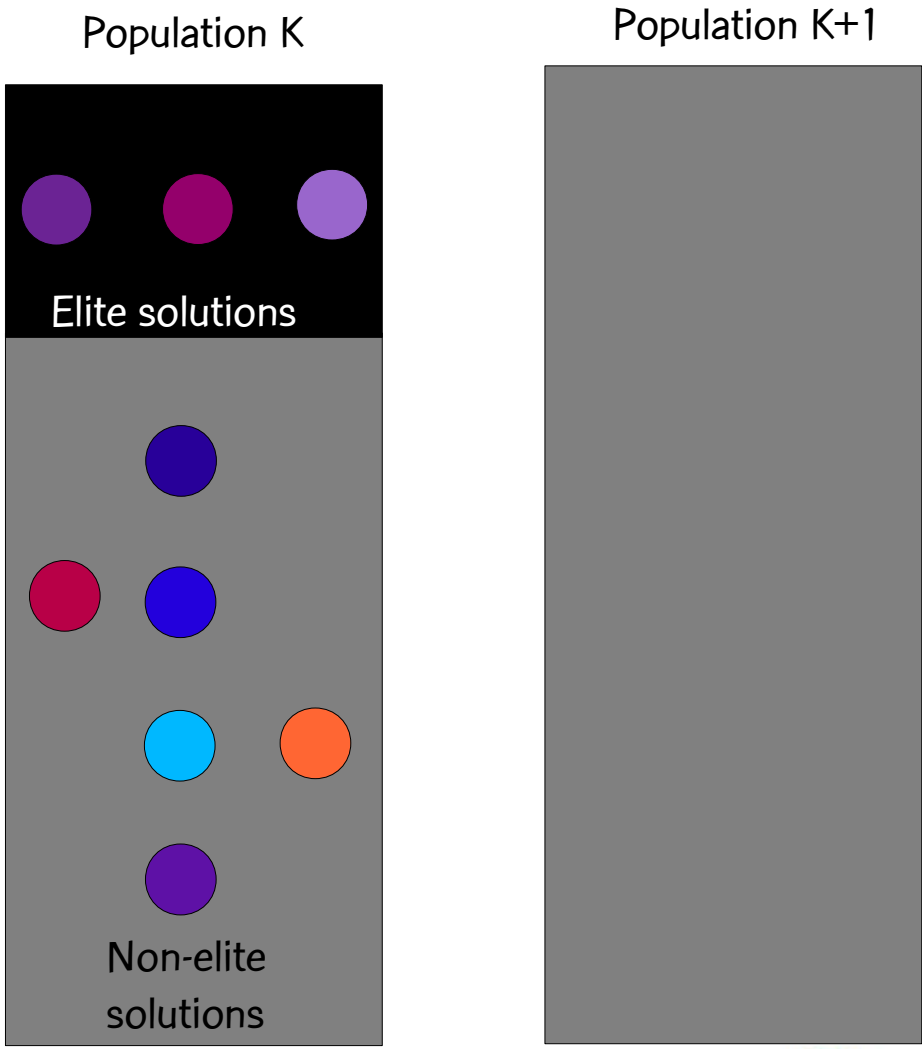
GAs and random keys

At the K-th generation, compute the cost of each solution and partition the solutions into two sets: elite solutions, non-elite solutions. Elite set should be smaller of the two sets and contain best solutions.



GAs and random keys

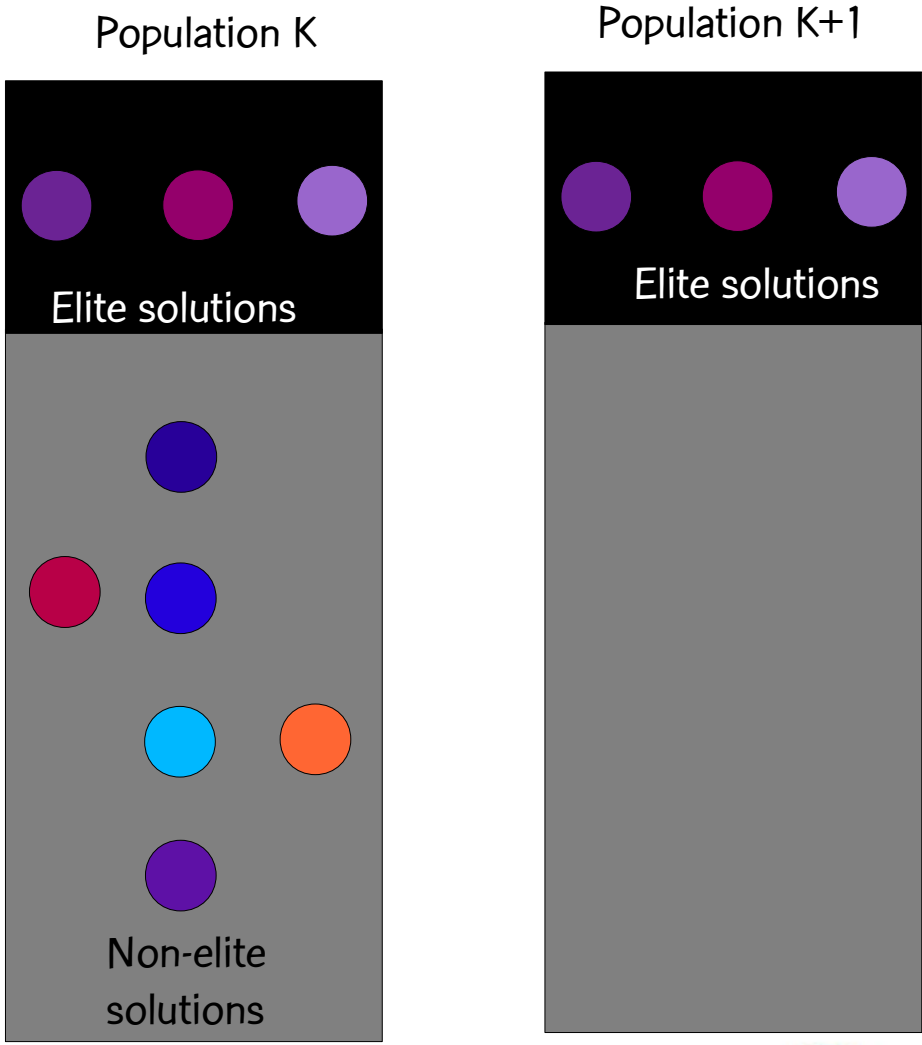
Evolutionary dynamics



GAs and random keys

Evolutionary dynamics

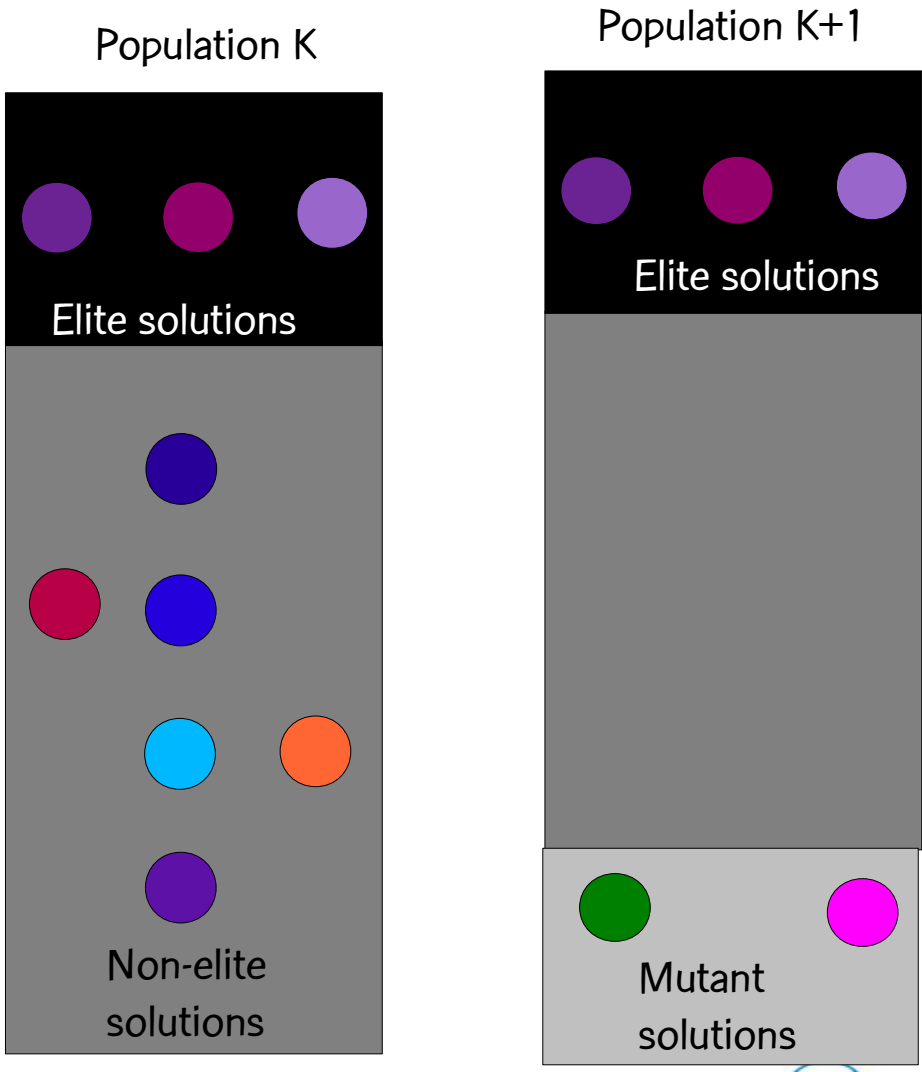
– Copy elite solutions from population K to population K+1



GAs and random keys

Evolutionary dynamics

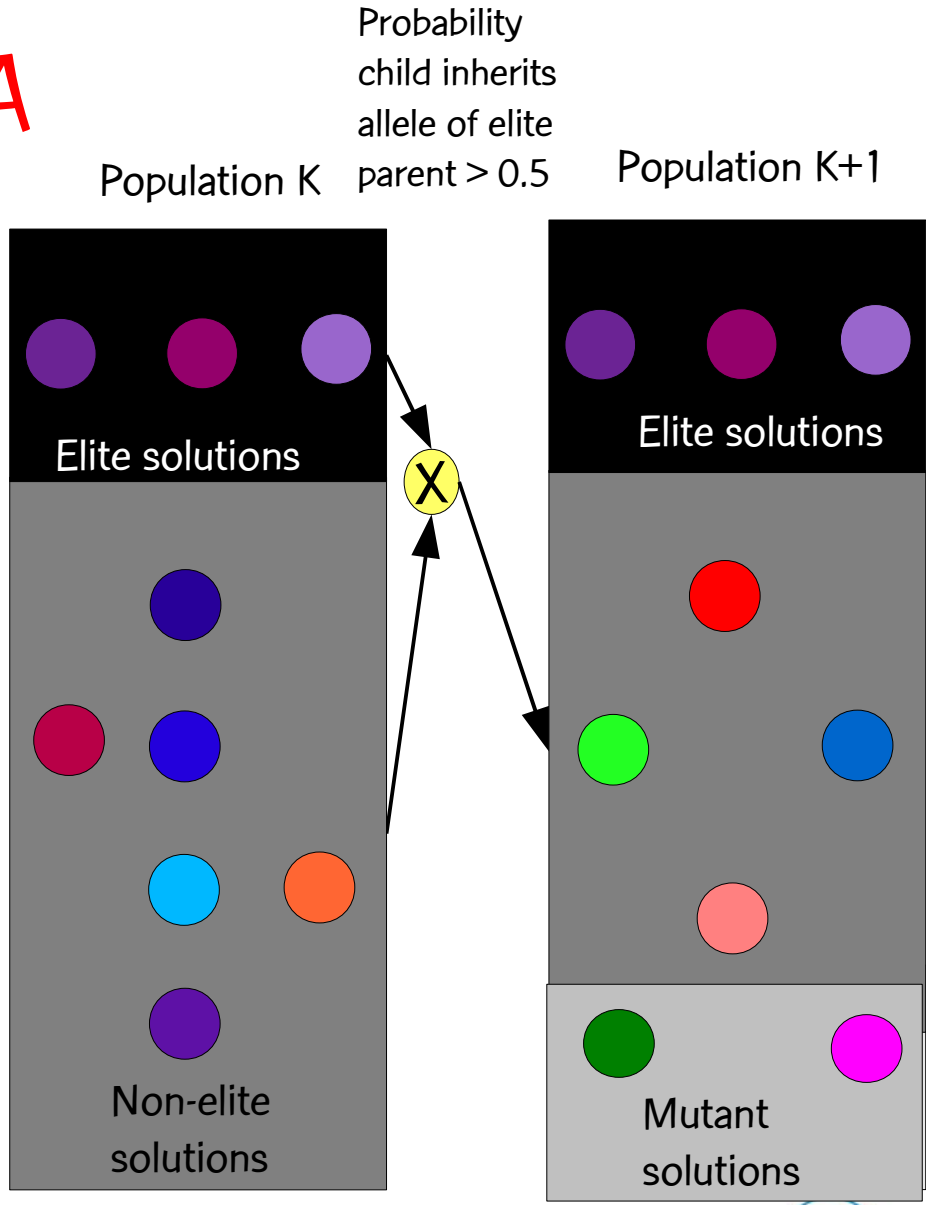
- Copy elite solutions from population K to population K+1
- Add R random solutions (mutants) to population K+1



Biased random key GA

Evolutionary dynamics

- Copy elite solutions from population K to population K+1
- Add R random solutions (mutants) to population K+1
- While K+1-th population < P
  - BIASED RANDOM KEY GA: Mate elite solution with non elite to produce child in population K+1. Mates are chosen at random.

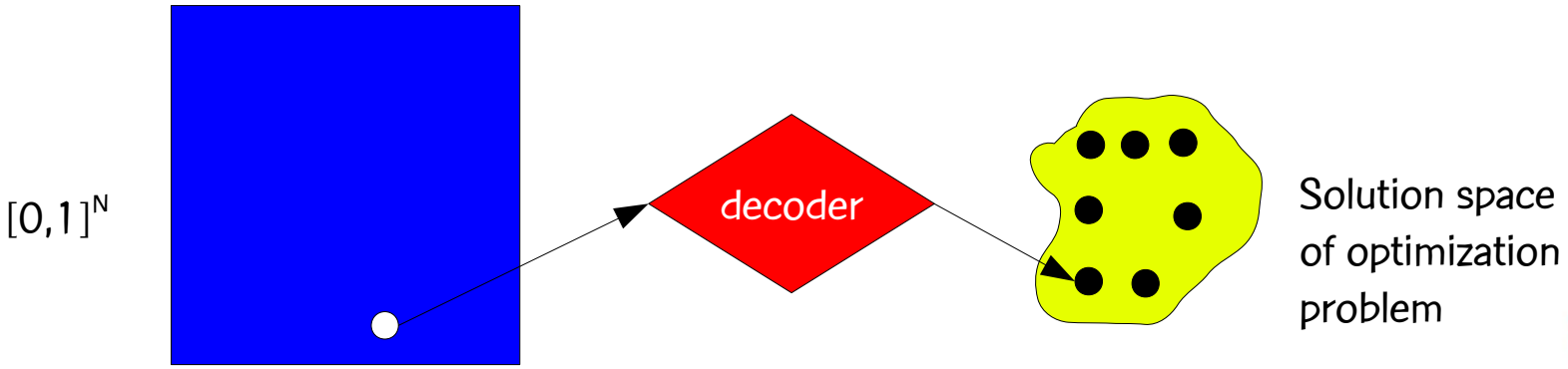


Observations

- Random method: keys are randomly generated so solutions are always random vectors
- Elitist strategy: best solutions are passed without change from one generation to the next
- Child inherits more characteristics of elite parent: one parent is always selected (with replacement) from the small elite set and probability that child inherits key of elite parent > 0.5
- No mutation in crossover: mutants are used instead

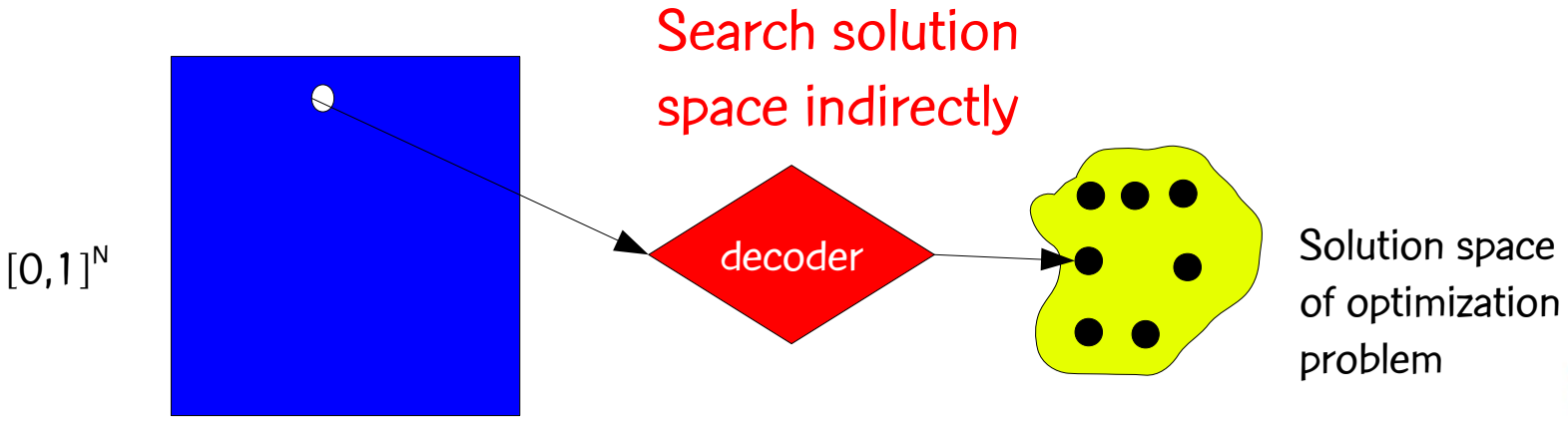
Decoders

- A decoder is a deterministic algorithm that takes as input a random-key vector and returns a feasible solution of the optimization problem and its cost.
- Bean (1994) proposed decoders based on sorting the random-key vector to produce a sequence.
- A random-key GA searches the solution space indirectly by searching the space of random keys and using the decoder to evaluate fitness of the random key.



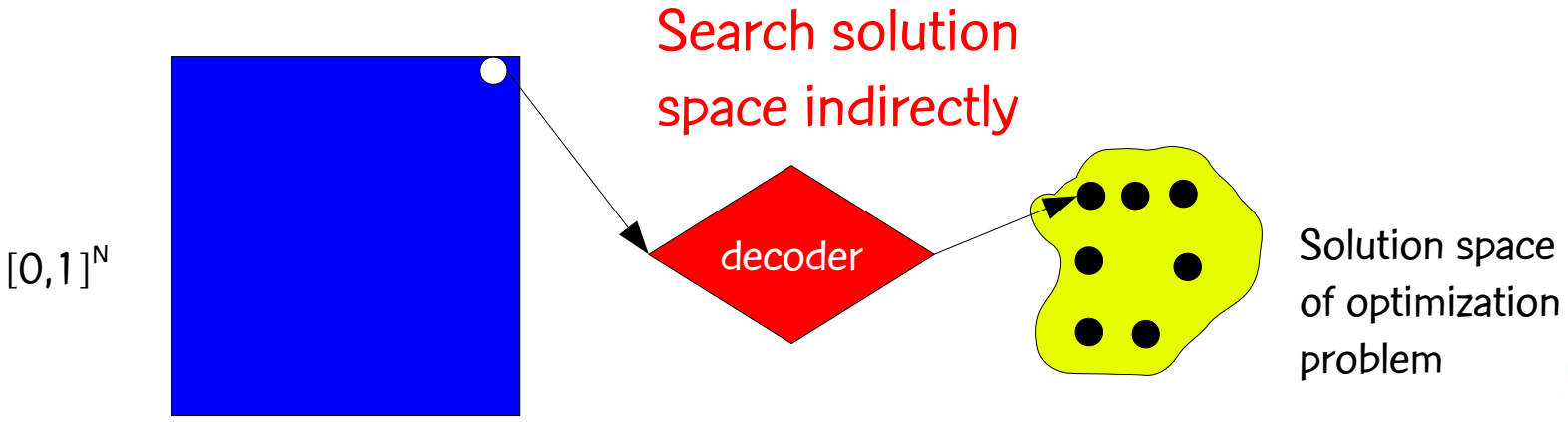
Decoders

- A decoder is a deterministic algorithm that takes as input a random-key vector and returns a feasible solution of the optimization problem and its cost.
- Bean (1994) proposed decoders based on sorting the random-key vector to produce a sequence.
- A random-key GA searches the solution space indirectly by searching the space of random keys and using the decoder to evaluate fitness of the random key.



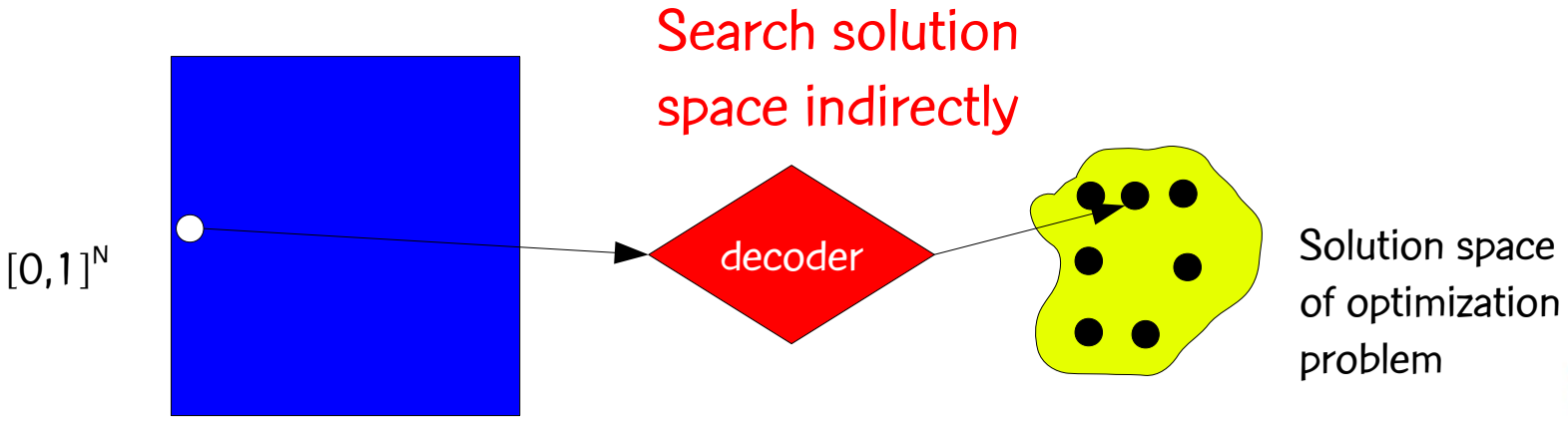
Decoders

- A decoder is a deterministic algorithm that takes as input a random-key vector and returns a feasible solution of the optimization problem and its cost.
- Bean (1994) proposed decoders based on sorting the random-key vector to produce a sequence.
- A random-key GA searches the solution space indirectly by searching the space of random keys and using the decoder to evaluate fitness of the random key.

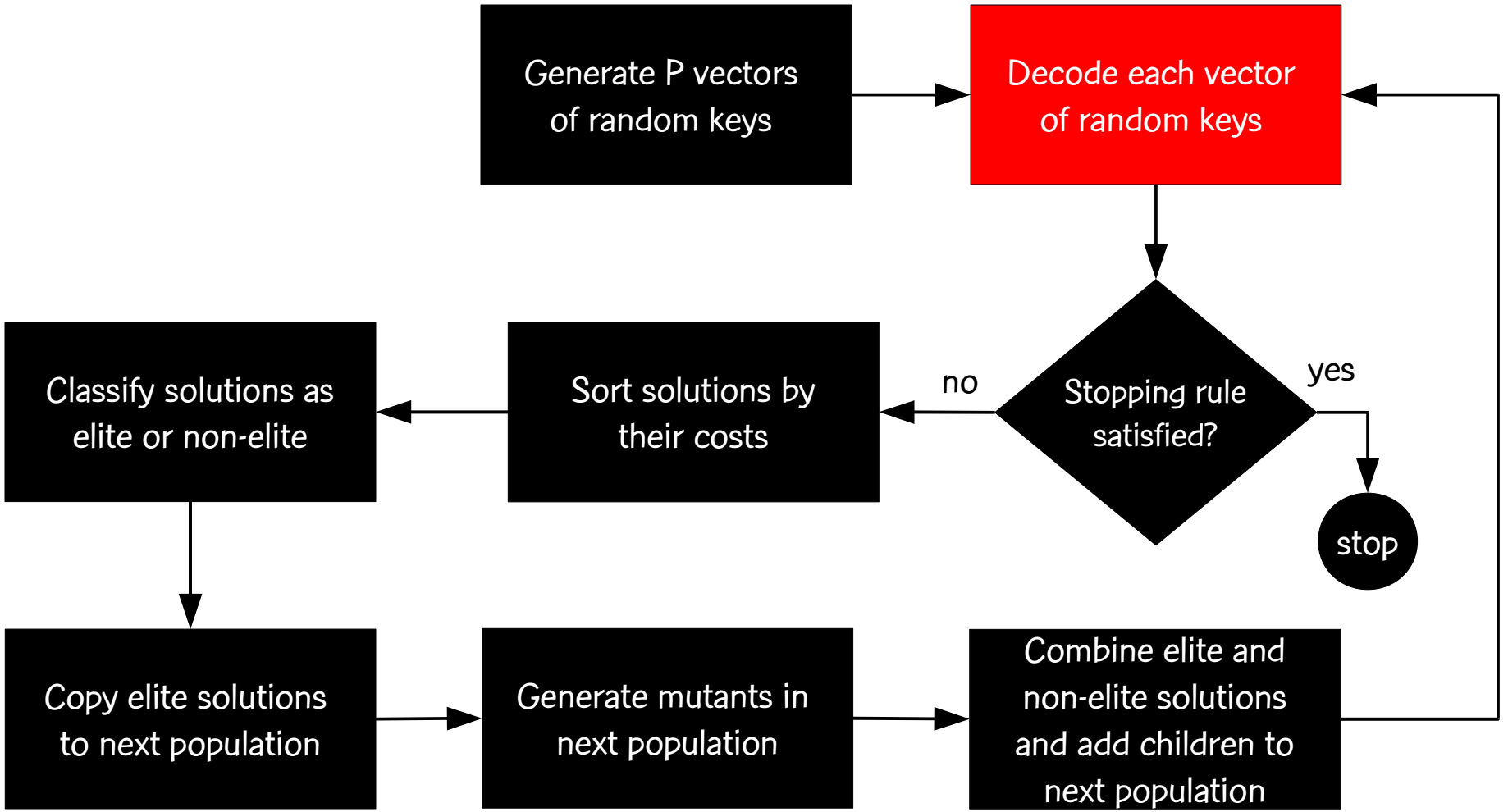


Decoders

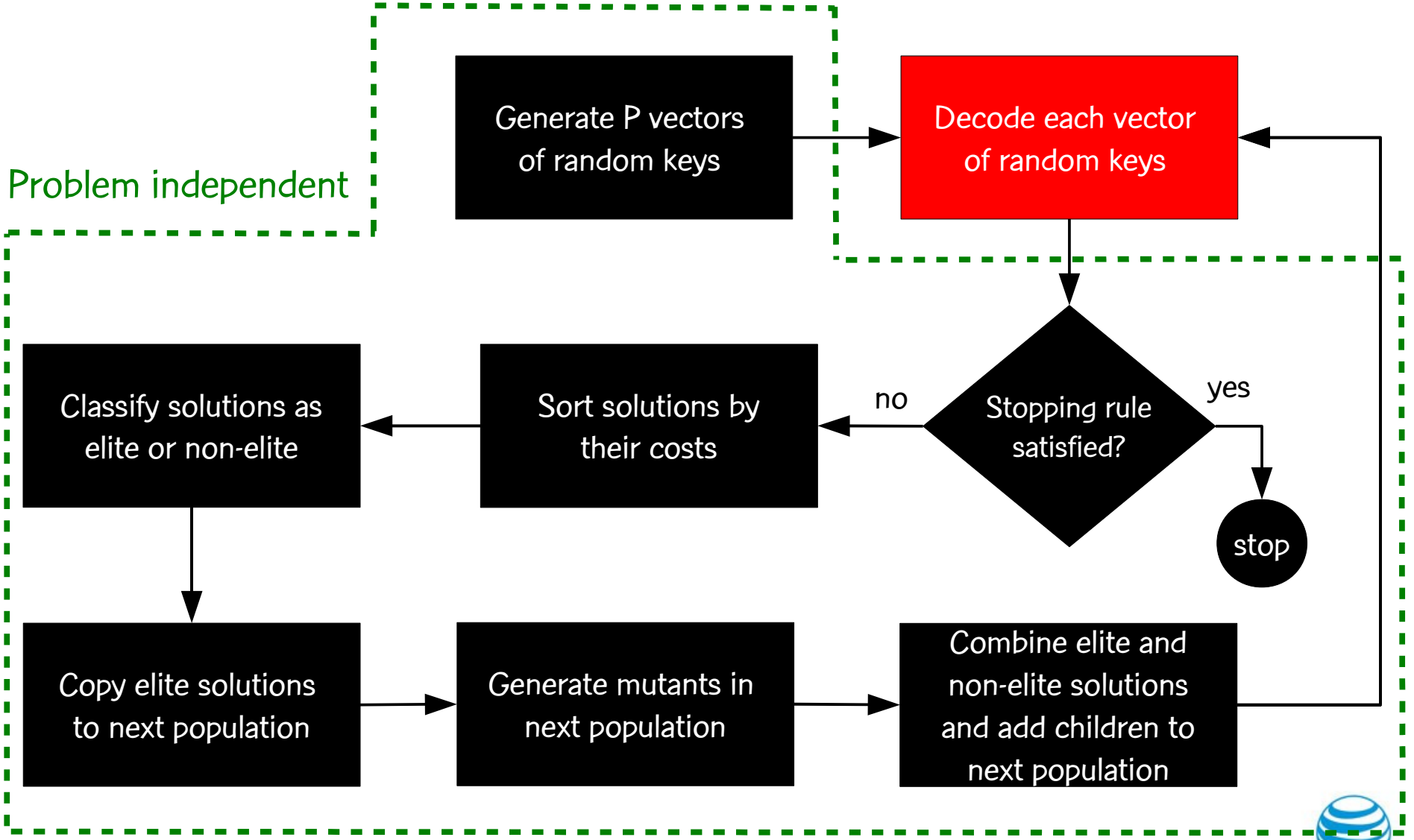
- A decoder is a deterministic algorithm that takes as input a random-key vector and returns a feasible solution of the optimization problem and its cost.
- Bean (1994) proposed decoders based on sorting the random-key vector to produce a sequence.
- A random-key GA searches the solution space indirectly by searching the space of random keys and using the decoder to evaluate fitness of the random key.



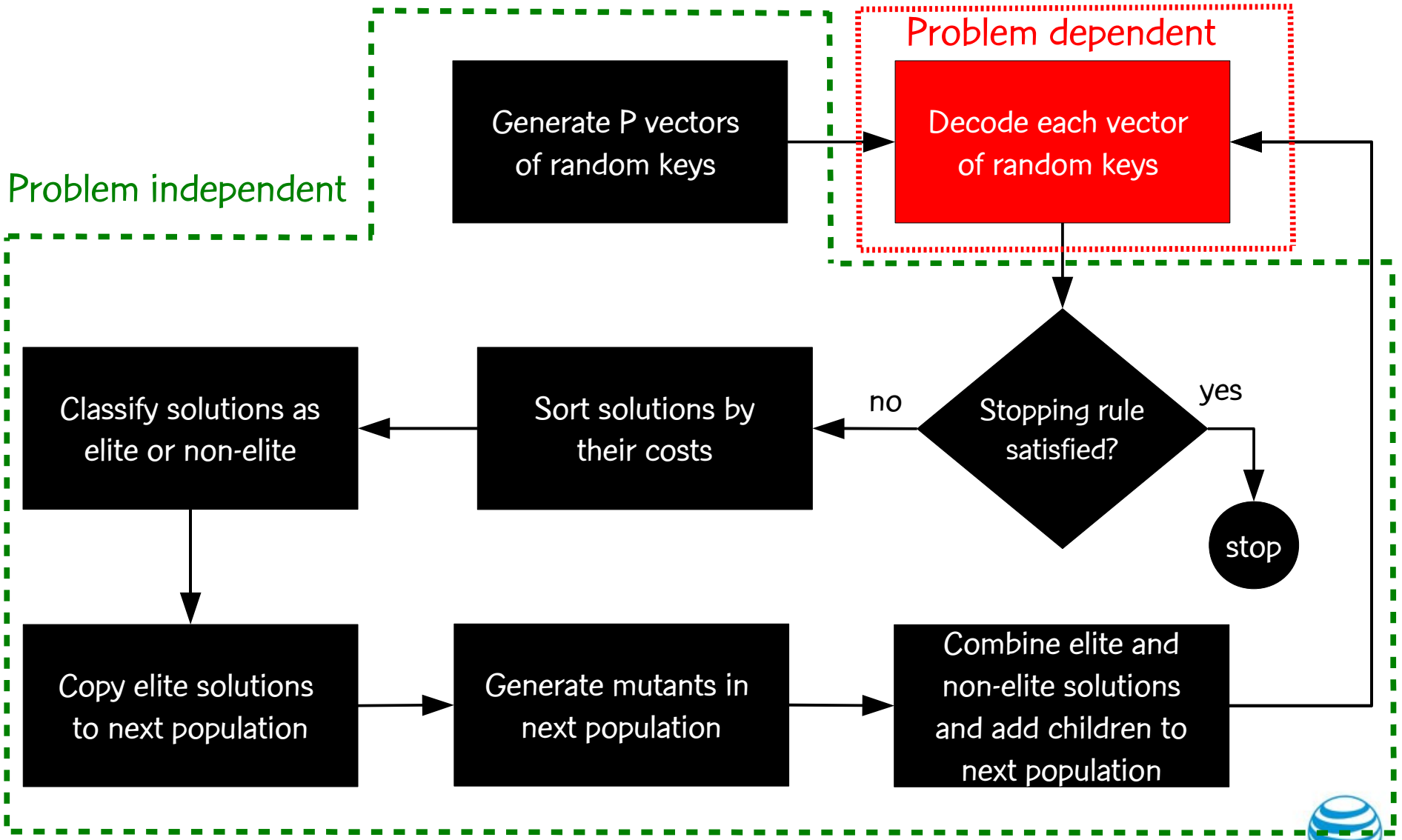
Framework for biased random-key genetic algorithms



Framework for biased random-key genetic algorithms



Framework for biased random-key genetic algorithms



Specifying a biased random-key GA

- Encoding is always done the same way, i.e. with a vector of N random-keys (parameter N must be specified)
- Decoder that takes as input a vector of N random-keys and outputs the corresponding solution of the combinatorial optimization problem and its cost (this is usually a heuristic)
- Parameters:
  - Size of population
  - Size of elite partition
  - Size of mutant set
  - Child inheritance probability
  - Stopping criterion

Specifying a biased random-key algorithm

- Encoding is always done the same way, i.e. with a vector of N random-keys (parameter N must be specified)
- Decoder that takes as input a vector of N random-keys and outputs the corresponding solution of the combinatorial optimization problem and its cost (this is usually a heuristic)
- Parameters:
  - Size of population: a function of N, say N or 2N
  - Size of elite partition
  - Size of mutant set
  - Child inheritance probability
  - Stopping criterion

Specifying a biased random-key algorithm

- Encoding is always done the same way, i.e. with a vector of N random-keys (parameter N must be specified)
- Decoder that takes as input a vector of N random-keys and outputs the corresponding solution of the combinatorial optimization problem and its cost (this is usually a heuristic)
- Parameters:
  - Size of population: a function of N, say N or 2N
  - Size of elite partition: 15-25% of population
  - Size of mutant set
  - Child inheritance probability
  - Stopping criterion

Specifying a biased random-key algorithm

- Encoding is always done the same way, i.e. with a vector of N random-keys (parameter N must be specified)
- Decoder that takes as input a vector of N random-keys and outputs the corresponding solution of the combinatorial optimization problem and its cost (this is usually a heuristic)
- Parameters:
  - Size of population: a function of N, say N or 2N
  - Size of elite partition: 15-25% of population
  - Size of mutant set: 5-15% of population
  - Child inheritance probability
  - Stopping criterion

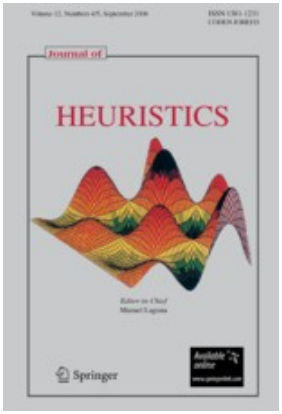
Specifying a biased random-key algorithm

- Encoding is always done the same way, i.e. with a vector of N random-keys (parameter N must be specified)
- Decoder that takes as input a vector of N random-keys and outputs the corresponding solution of the combinatorial optimization problem and its cost (this is usually a heuristic)
- Parameters:
  - Size of population: a function of N, say N or 2N
  - Size of elite partition: 15-25% of population
  - Size of mutant set: 5-15% of population
  - Child inheritance probability: > 0.5, say 0.7
  - Stopping criterion

Specifying a biased random-key algorithm

- Encoding is always done the same way, i.e. with a vector of N random-keys (parameter N must be specified)
- Decoder that takes as input a vector of N random-keys and outputs the corresponding solution of the combinatorial optimization problem and its cost (this is usually a heuristic)
- Parameters:
  - Size of population: a function of N, say N or 2N
  - Size of elite partition: 15-25% of population
  - Size of mutant set: 5-15% of population
  - Child inheritance probability: > 0.5, say 0.7
  - Stopping criterion: e.g. time, # generations, solution quality, # generations without improvement

Reference



J.F. Gonçalves and M.G.C.R., "Biased random-key genetic algorithms for combinatorial optimization," J. of Heuristics, published online 31 August 2010.

Tech report version:

http://www2.research.att.com/~mgcr/doc/srkga.pdf

BRKGA for Steiner triple covering

Encoding a solution to a vector of random keys

A solution is encoded as an n-vector $X = (X_1, X_2, \ldots, X_n)$ of random keys where n is the number of columns of matrix A.

Encoding a solution to a vector of random keys

A solution is encoded as an n-vector $X = (X_1, X_2, \ldots, X_n)$ of random keys where n is the number of columns of matrix A.

Each key is a randomly generated number in the real interval [0,1).

Encoding a solution to a vector of random keys

A solution is encoded as an n-vector $X = (X_1, X_2, \ldots, X_n)$ of random keys where n is the number of columns of matrix A.

Each key is a randomly generated number in the real interval [0,1).

The j-th component of X corresponds to the j-th column of A.

Decoding a solution from a vector of random keys

Decoder takes as input an n-vector $X = (X_1, X_2, ..., X_n)$ of random keys and returns a cover $J^* \subseteq \{1, 2, ..., n\}$ corresponding to the indices of the columns of A selected to cover the rows of A.

Decoding a solution from a vector of random keys

Decoder takes as input an n-vector $X = (X_1, X_2, ..., X_n)$ of random keys and returns a cover $J^* \subseteq \{1, 2, ..., n\}$ corresponding to the indices of the columns of A selected to cover the rows of A.

Let $Y = (Y_1, Y_2, ..., Y_n)$ be a binary vector where $Y_j = 1$ if and only if $j \in J^*$.

Decoding a solution from a vector of random keys

Decoder has three phases:

Phase I: For $j = 1, 2, \ldots, n$, set $Y_j = 1$ if $X_j \geq ½$, set $Y_j = 0$ otherwise.

Decoding a solution from a vector of random keys

Decoder has three phases:

Phase I: For $j = 1, 2, \ldots, n$, set $Y_j = 1$ if $X_j \geq ½$, set $Y_j = 0$ otherwise.

The indices implied by the binary vector can correspond to either a feasible or infeasible cover.

If cover is feasible, Phase II is skipped.

Decoding a solution from a vector of random keys

Decoder has three phases:

Phase II: If J* is not a valid cover, build a cover with a greedy algorithm for set covering (Johnson, 1974) starting from the partial cover J*.

Decoding a solution from a vector of random keys

Decoder has three phases:

Phase II: If J* is not a valid cover, build a cover with a greedy algorithm for set covering (Johnson, 1974) starting from the partial cover J*.

Greedy algorithm: While J* is not a valid cover, select to add in J* the smallest index $j \in \{1,2,\ldots,n\} \setminus J^*$ for which the inclusion of j in J* covers the maximum number of yet-uncovered rows.

Decoding a solution from a vector of random keys

Decoder has three phases:

Phase III: Local search attempts to remove superfluous columns from cover J*.

Decoding a solution from a vector of random keys

Decoder has three phases:

Phase III: Local search attempts to remove superfluous columns from cover J*.

Local search: While there is some element $j \in J^*$ such that $J^* \setminus \{ j \}$ is still a valid cover, then such element having the smallest index is removed from J*.

Implementation issues

Implementation issues

- BRKGA framework (R. and Toso, 2010), a C++ framework for biased random-key genetic algorithms.
  - Object oriented
  - Multi-threaded: parallel decoding using OpenMP
  - General-purpose framework: implements all problem independent components and provides a simple hook for chromosome decoding
  - Chromosome correcting

Implementation issues

Chromosome correcting: decoder not only returns the cover J* but also modifies the vector X of random keys such that it decodes directly into J* with the application of only the first phase of the decoder:

Implementation issues

Chromosome correcting: decoder not only returns the cover J* but also modifies the vector X of random keys such that it decodes directly into J* with the application of only the first phase of the decoder:

X_j is unchanged if $X_j \geq 1/2$ and $j \in J^*$ or if $X_j < 1/2$ and $j \notin J^*$

X_j changes to $1 - X_j$ if $X_j < 1/2$ and $j \in J^*$ or if $X_j \geq 1/2$ and $j \notin J^*$

Experimental results

Experiments: objectives

- Investigate effectiveness of BRKGA to find optimal covers for instances with known optimum.

Experiments: objectives

- Investigate effectiveness of BRKGA to find optimal covers for instances with known optimum.
- For the two instances (stn405 and stn729) for which optimal solutions are not known, attempt to produce better covers than previously found.

Experiments: objectives

- Investigate effectiveness of BRKGA to find optimal covers for instances with known optimum.
- For the two instances (stn405 and stn729) for which optimal solutions are not known, attempt to produce better covers than previously found.
- Investigate effectiveness of parallel implementation.

Experiments: instances

We use the set of instances: stn9, stn15, stn27, stn45, stn81, stn135, stn243, stn405, stn729

Instances can be downloaded from:

http://www2.research.att.com/~mgcr/data/steiner-triple-covering.tar.gz

Experiments: computing environment

Computer: server with four 2.4 GHz Quad-core Intel Xeon E7330 processors with 128 Gb of memory, running CentOS 5 Linux. Total of 16 processors.

Experiments: computing environment

Computer: server with four 2.4 GHz Quad-core Intel Xeon E7330 processors with 128 Gb of memory, running CentOS 5 Linux. Total of 16 processors.

Compiler: g++ version 4.1.2 20080704 with flags -O3 -fopenmp

Experiments: computing environment

Computer: server with four 2.4 GHz Quad-core Intel Xeon E7330 processors with 128 Gb of memory, running CentOS 5 Linux. Total of 16 processors.

Compiler: g++ version 4.1.2 20080704 with flags -O3 -fopenmp

Random number generator: Mersenne Twister (Matsumoto & Nishimura, 1998)

Experiments: multi-population GA

We evolve 3 populations simultaneously (but sequentially).

Experiments: multi-population GA

We evolve 3 populations simultaneously (but sequentially).

Every 100 generations the best two solutions from each population replaces the worst solutions of the other two populations if not already present there.

Experiments: multi-population GA

We evolve 3 populations simultaneously (but sequentially).

Every 100 generations the best two solutions from each population replaces the worst solutions of the other two populations if not already present there.

Parallel processing is only done when calling the decoder. Up to 16 chromosomes are decoded in parallel.

Experiments: other parameters

Population size: 10n, where n is the number of columns of A

Experiments: other parameters

Population size: $10n$, where n is the number of columns of A

Population partition: $\lceil 1.5n \rceil$ elite solutions; $1-\lceil 1.5n \rceil = \lfloor 8.5n \rfloor$ non-elite solutions

Experiments: other parameters

Population size: $10n$, where n is the number of columns of A

Population partition: $\lceil 1.5n \rceil$ elite solutions; $1-\lceil 1.5n \rceil = \lfloor 8.5n \rfloor$ non-elite solutions

Mutants: $\lfloor 5.5n \rfloor$ are created at each generation

Experiments: other parameters

Population size: $10n$, where n is the number of columns of A

Population partition: $\lceil 1.5n \rceil$ elite solutions; $1-\lceil 1.5n \rceil = \lfloor 8.5n \rfloor$ non-elite solutions

Mutants: $\lfloor 5.5n \rfloor$ are created at each generation

Probability child inherits gene of elite/non-elite parent: biased coin 60% : 40%

Experiments: other parameters

Population size: $10n$, where n is the number of columns of A

Population partition: $\lceil 1.5n \rceil$ elite solutions; $1-\lceil 1.5n \rceil = \lfloor 8.5n \rfloor$ non-elite solutions

Mutants: $\lfloor 5.5n \rfloor$ are created at each generation

Probability child inherits gene of elite/non-elite parent: biased coin 60% : 40%

Stopping rule: we use different stopping rules for each of the three types of experiments

Experiments on instances with known optimal covers

For each instance: ran *GA* independently 100 times, stopping when an optimal cover was found.

Experiments on instances with known optimal covers

For each instance: ran *GA* independently 100 times, stopping when an optimal cover was found.

On all 100 runs for each instance, the algorithm found an optimal cover.

Experiments on instances with known optimal covers

For each instance: ran GA independently 100 times, stopping when an optimal cover was found.

On all 100 runs for each instance, the algorithm found an optimal cover.

On the smallest instances (stn9, stn15, stn27) an optimal cover was always found in the initial population.

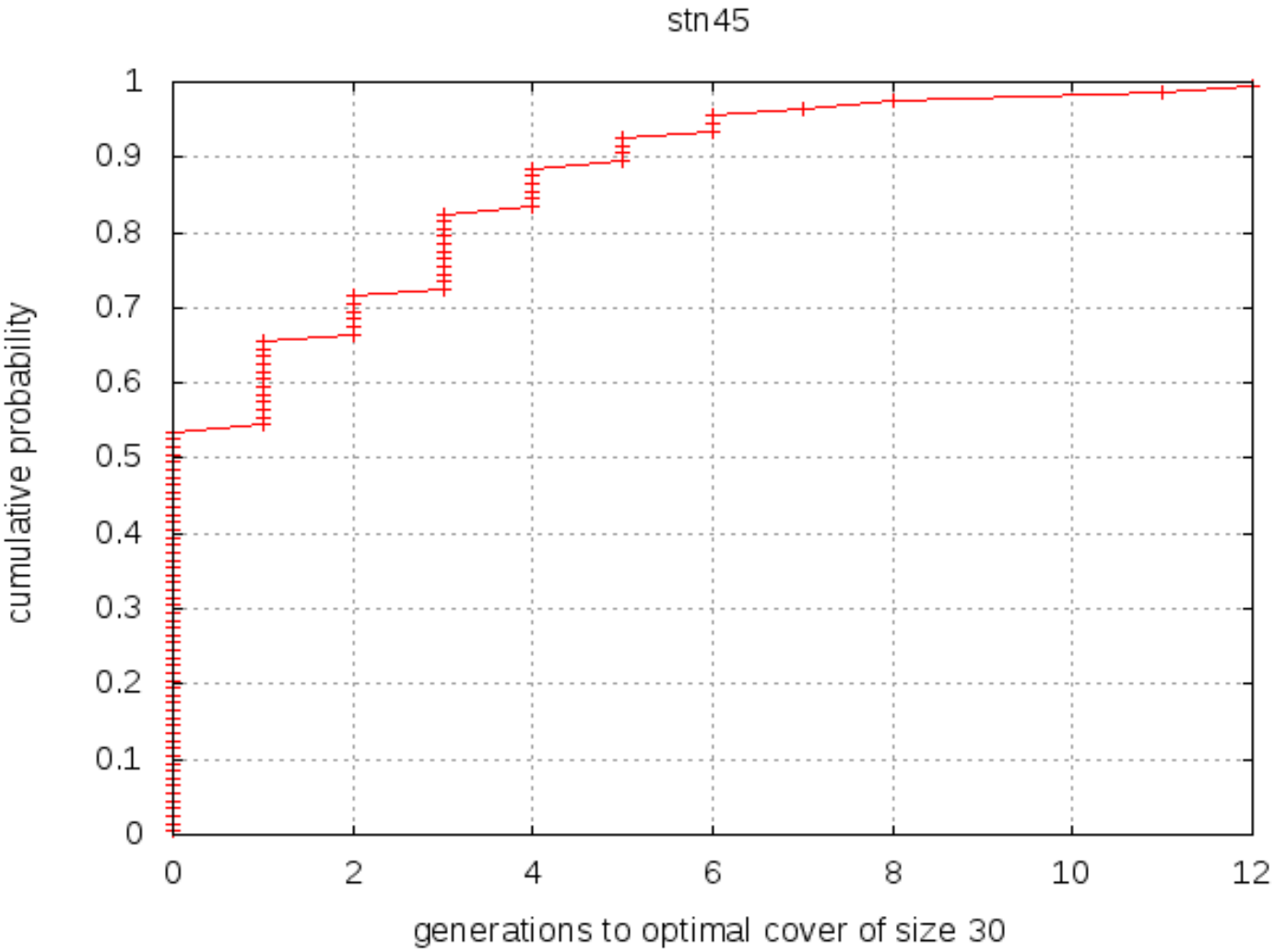
Experiments on instances with known optimal covers

For each instance: ran GA independently 100 times, stopping when an optimal cover was found.

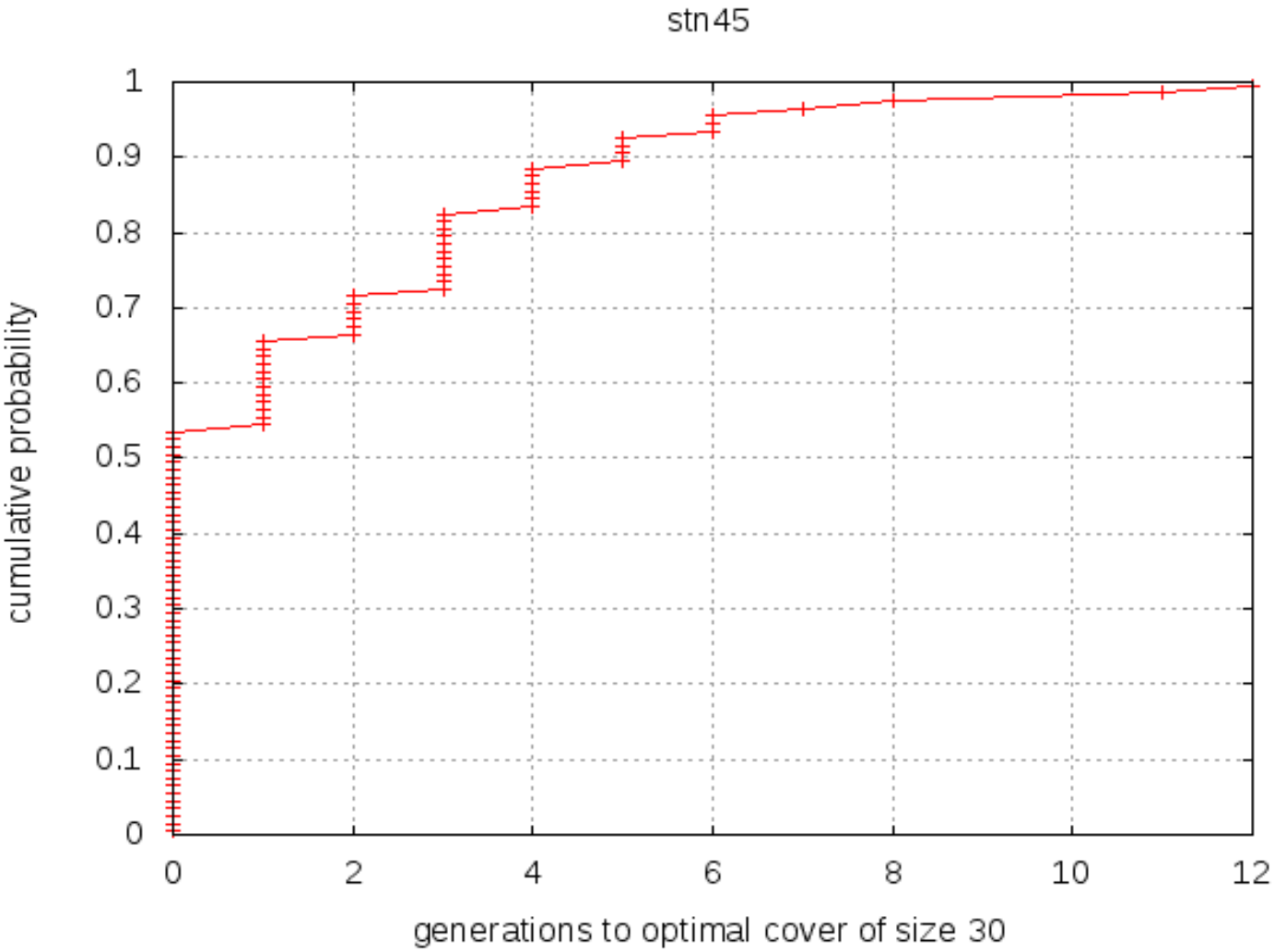
On all 100 runs for each instance, the algorithm found an optimal cover.

On the smallest instances (stn9, stn15, stn27) an optimal cover was always found in the initial population.

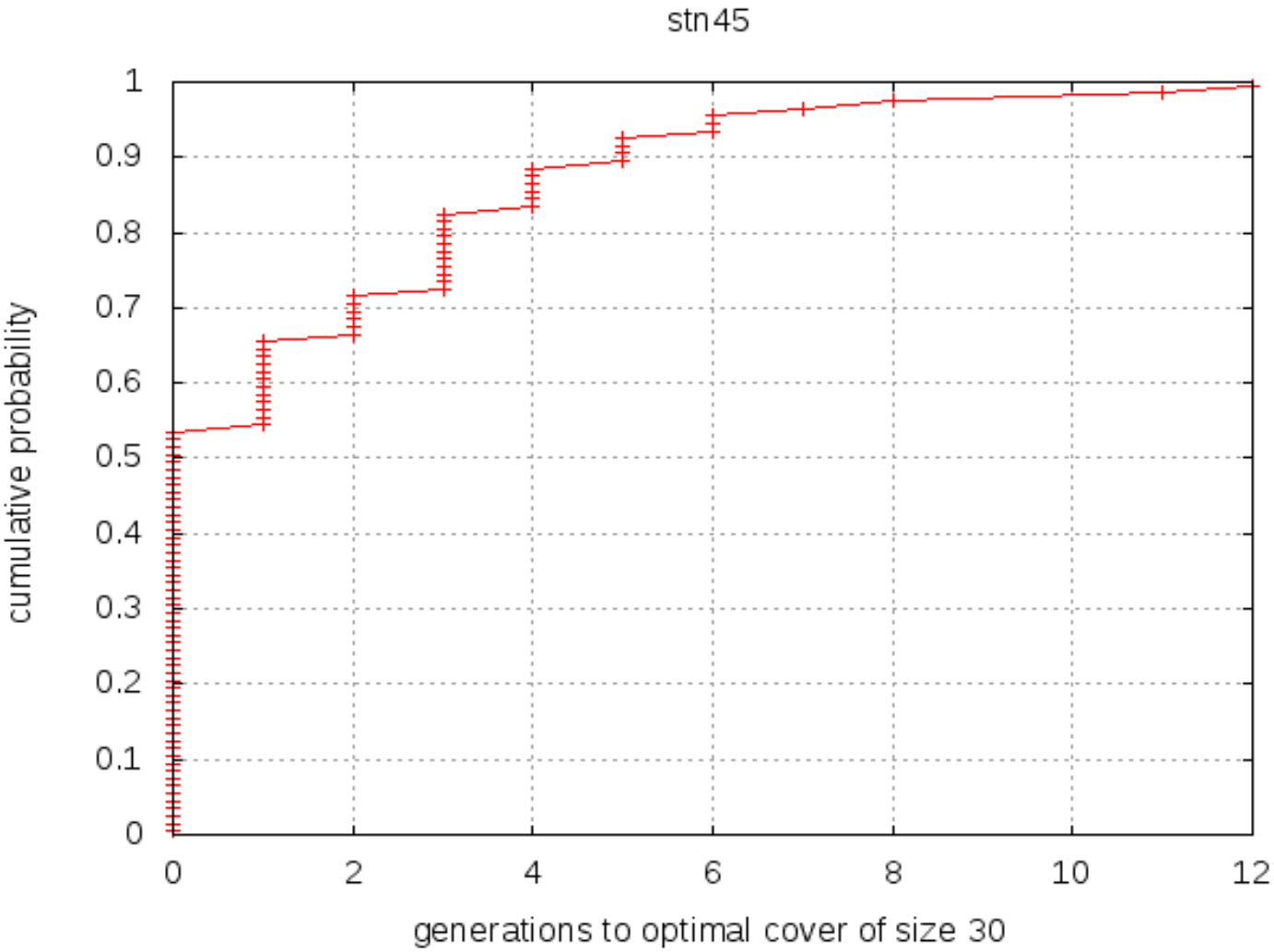
On stn81 an optimal cover was found in the initial population in 99 of the 100 runs. In the remaining run, an optimal cover was found in the second iteration.



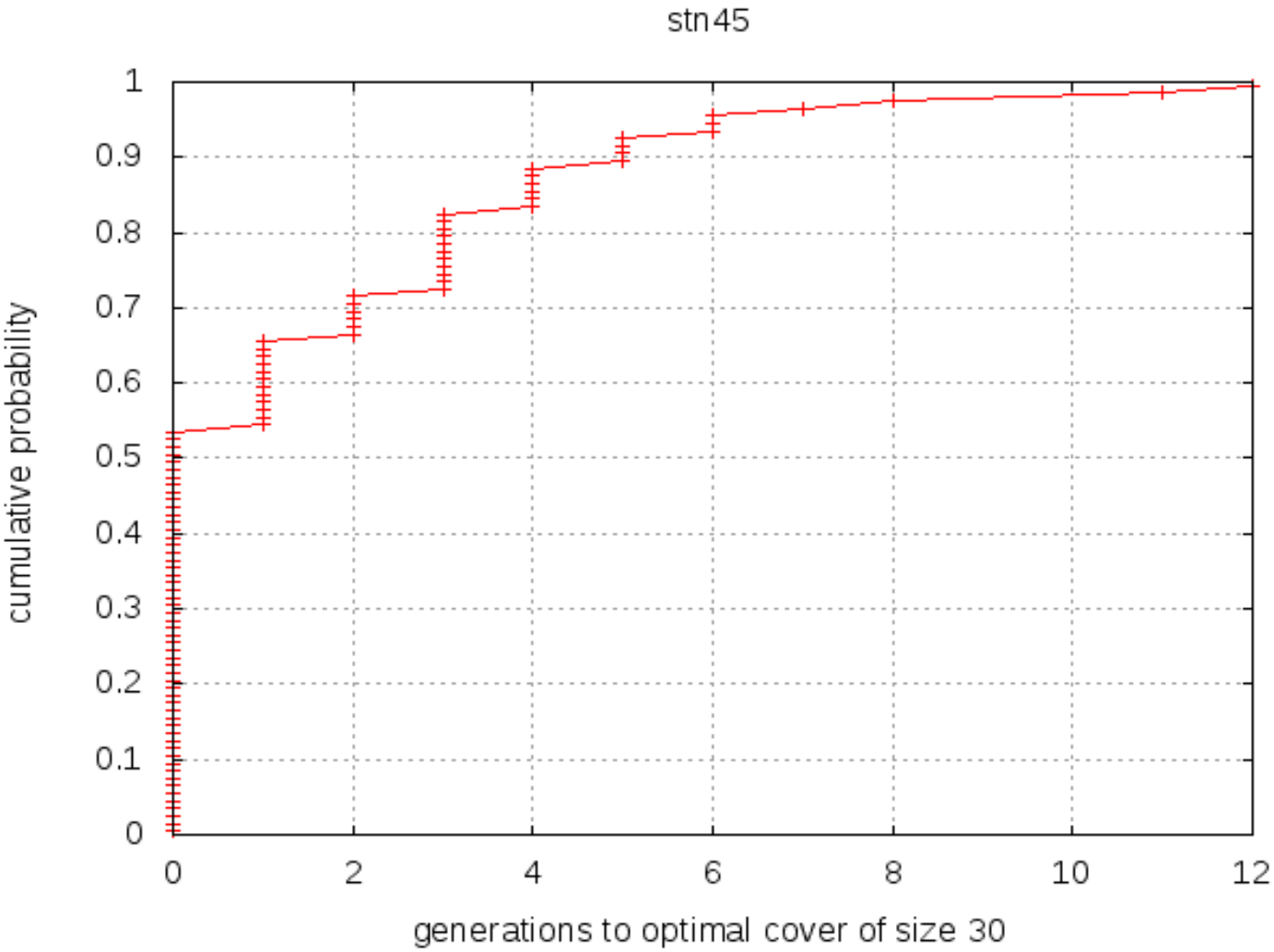
Instance stn45



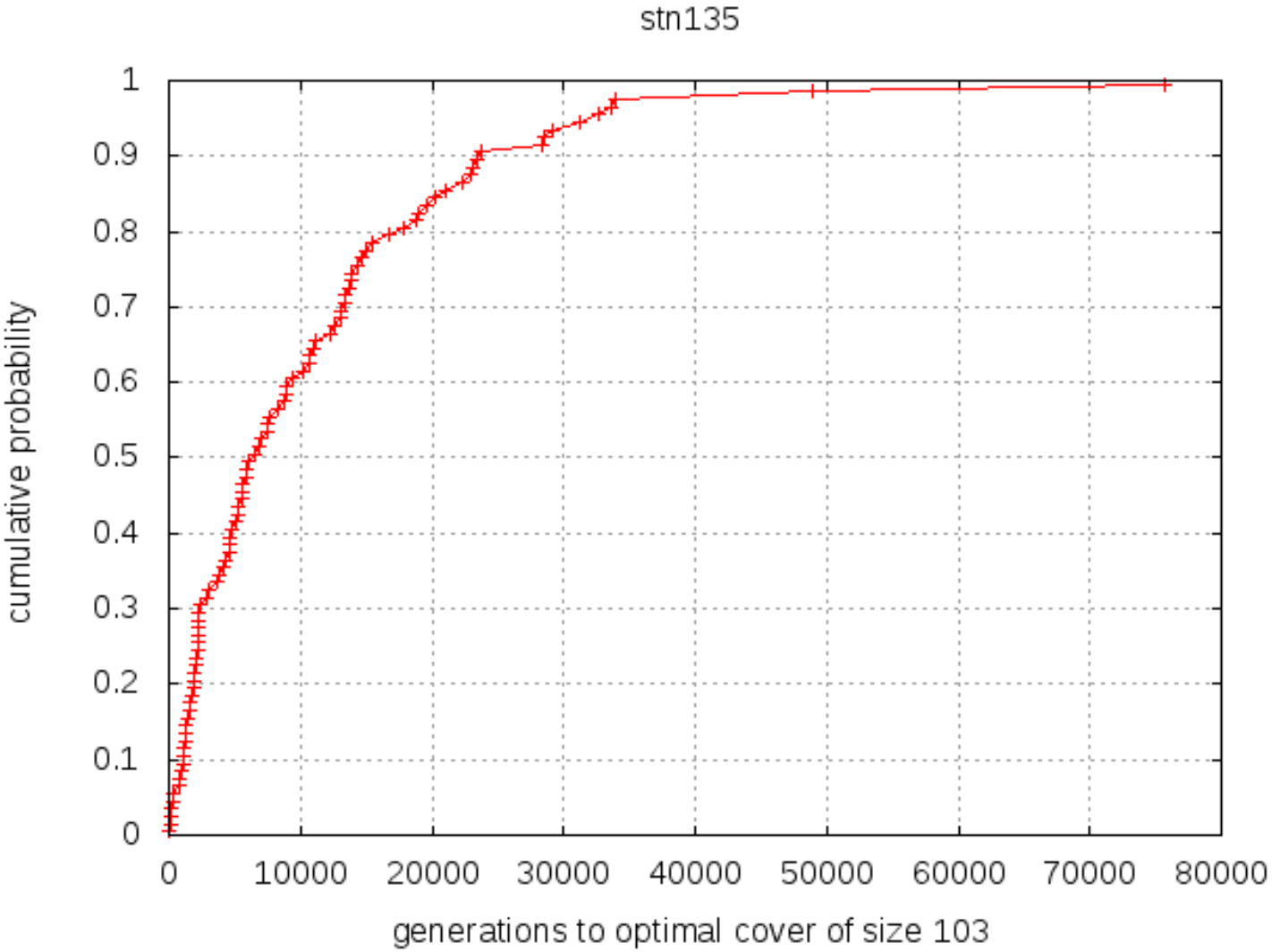
Optimal cover found in initial population in 54/100 runs



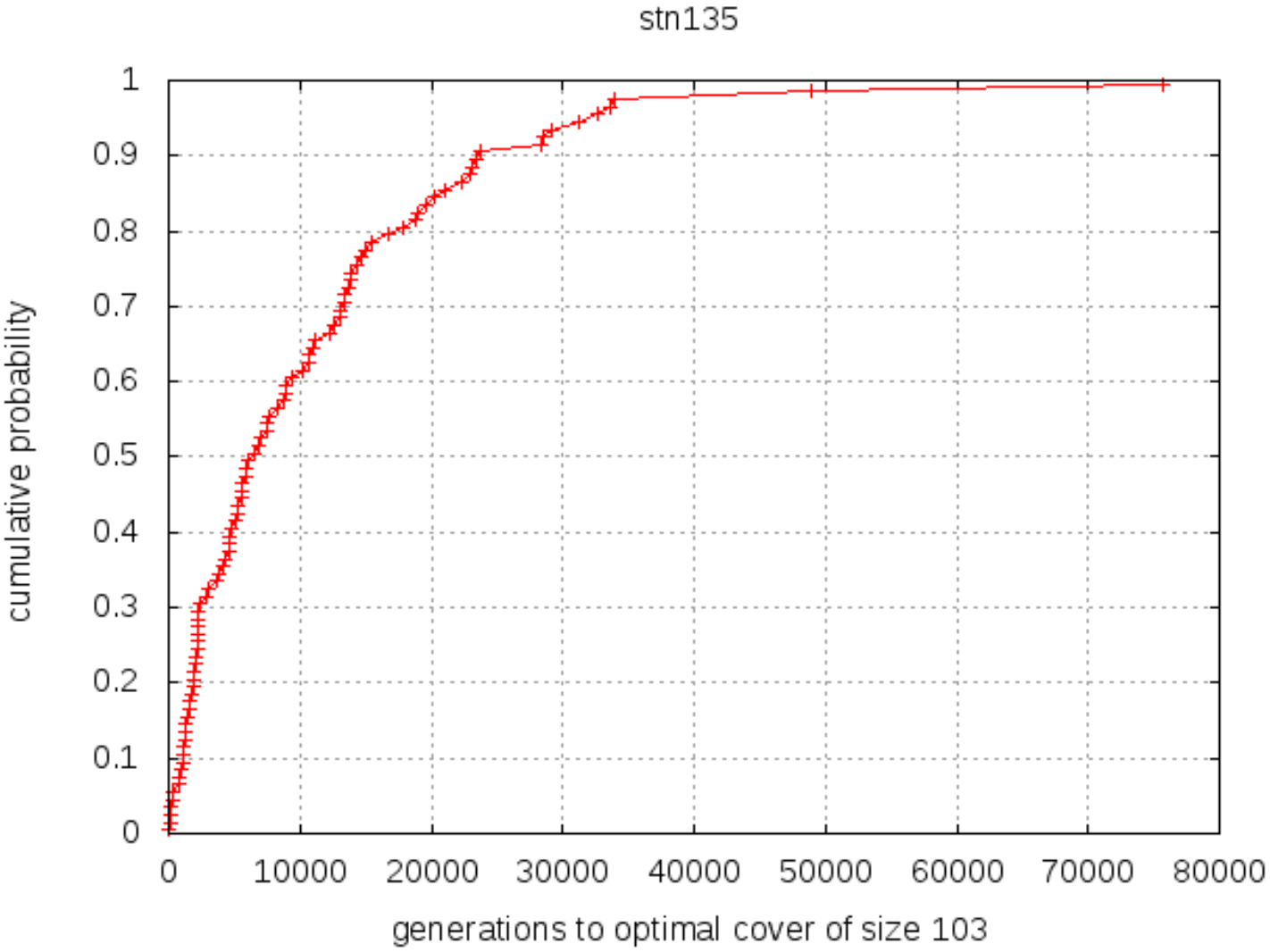
Largest number of iterations in 100 runs was 12



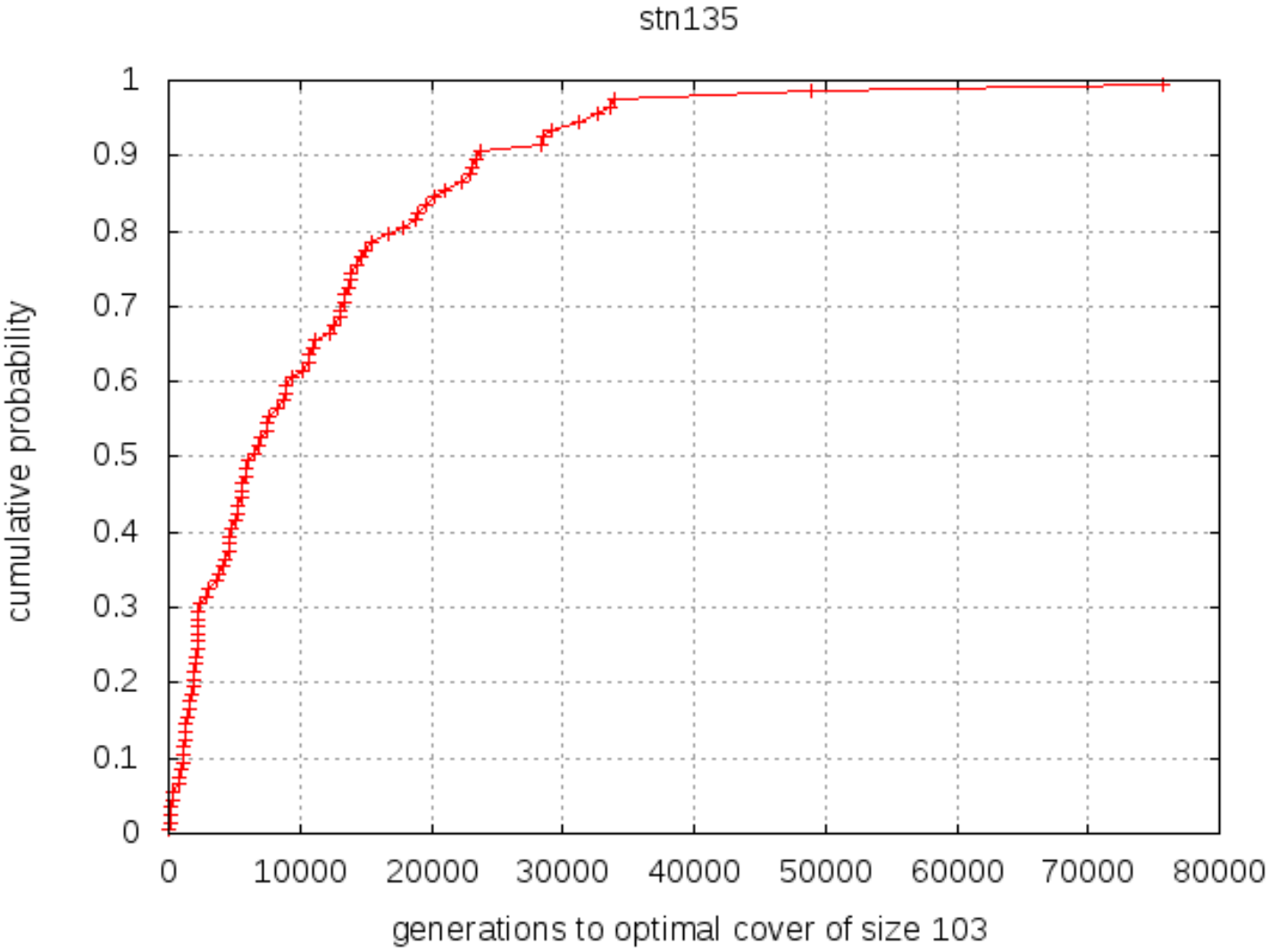
Time per 1000 generations: 4.70s (real), 70.55s (user), 2.73s (sys)



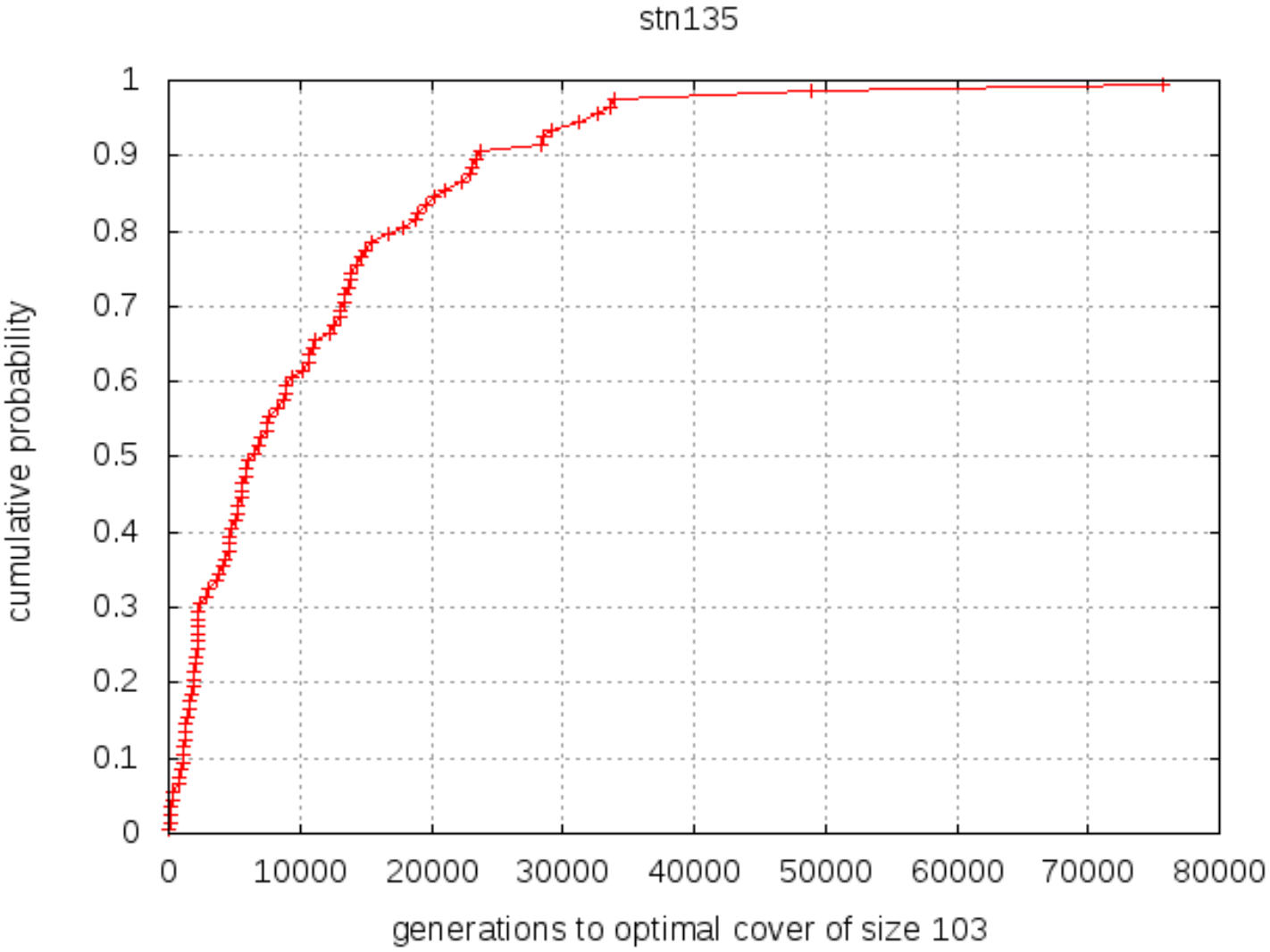
Instance stn135



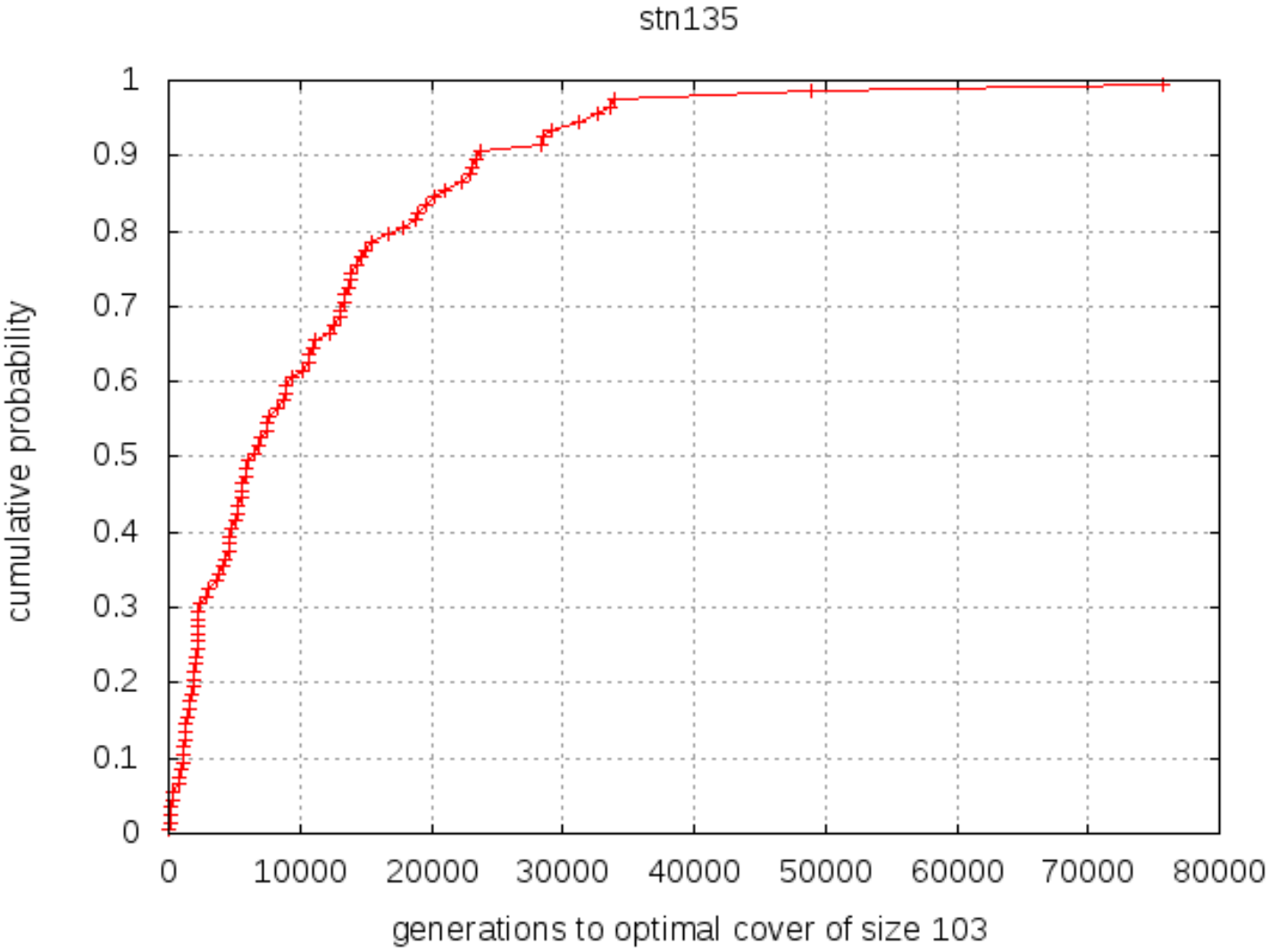
Most difficult instance of those with known optimal cover



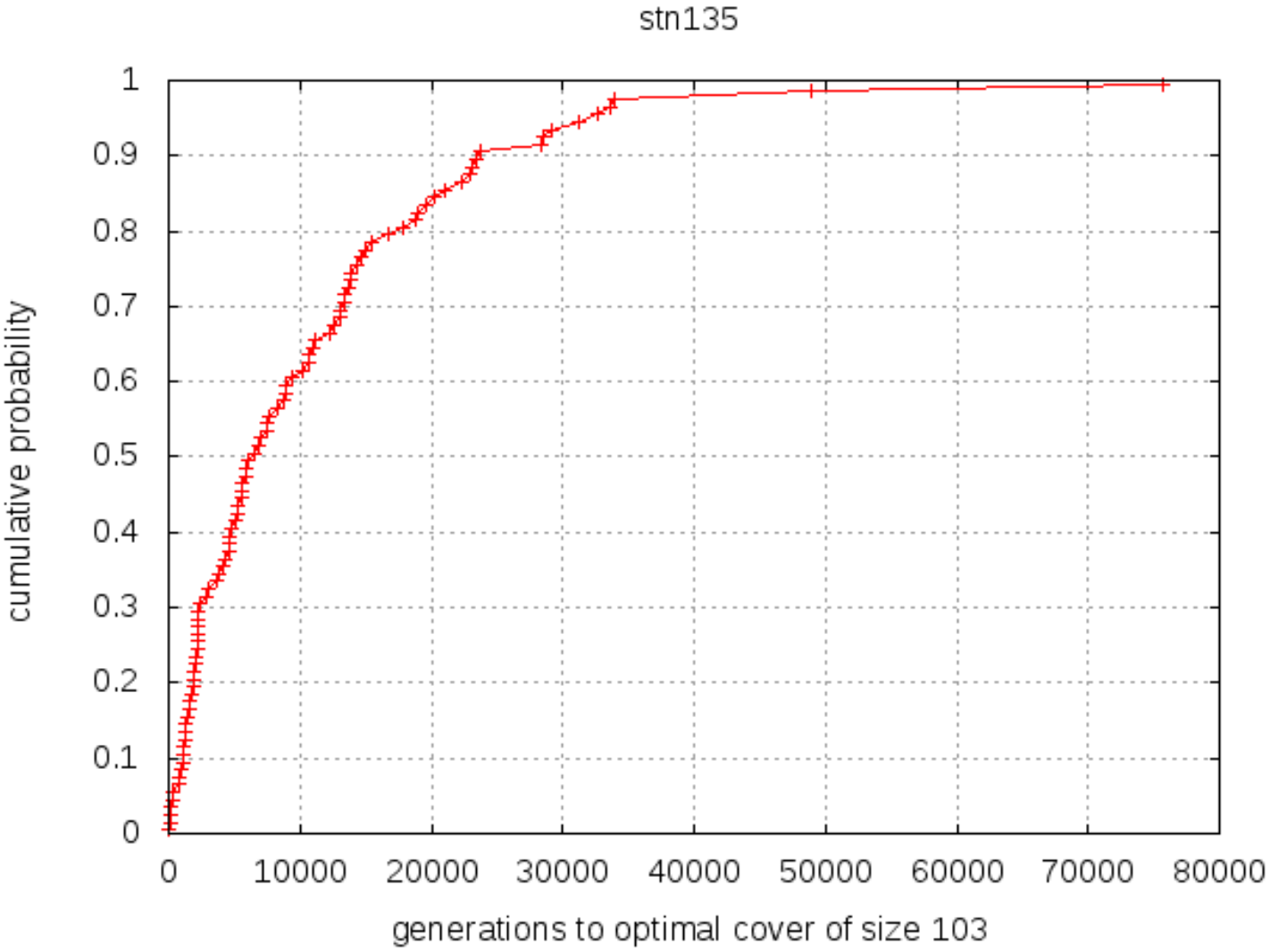
9 of the 100 runs found an optimal cover in less than 1000 iterations



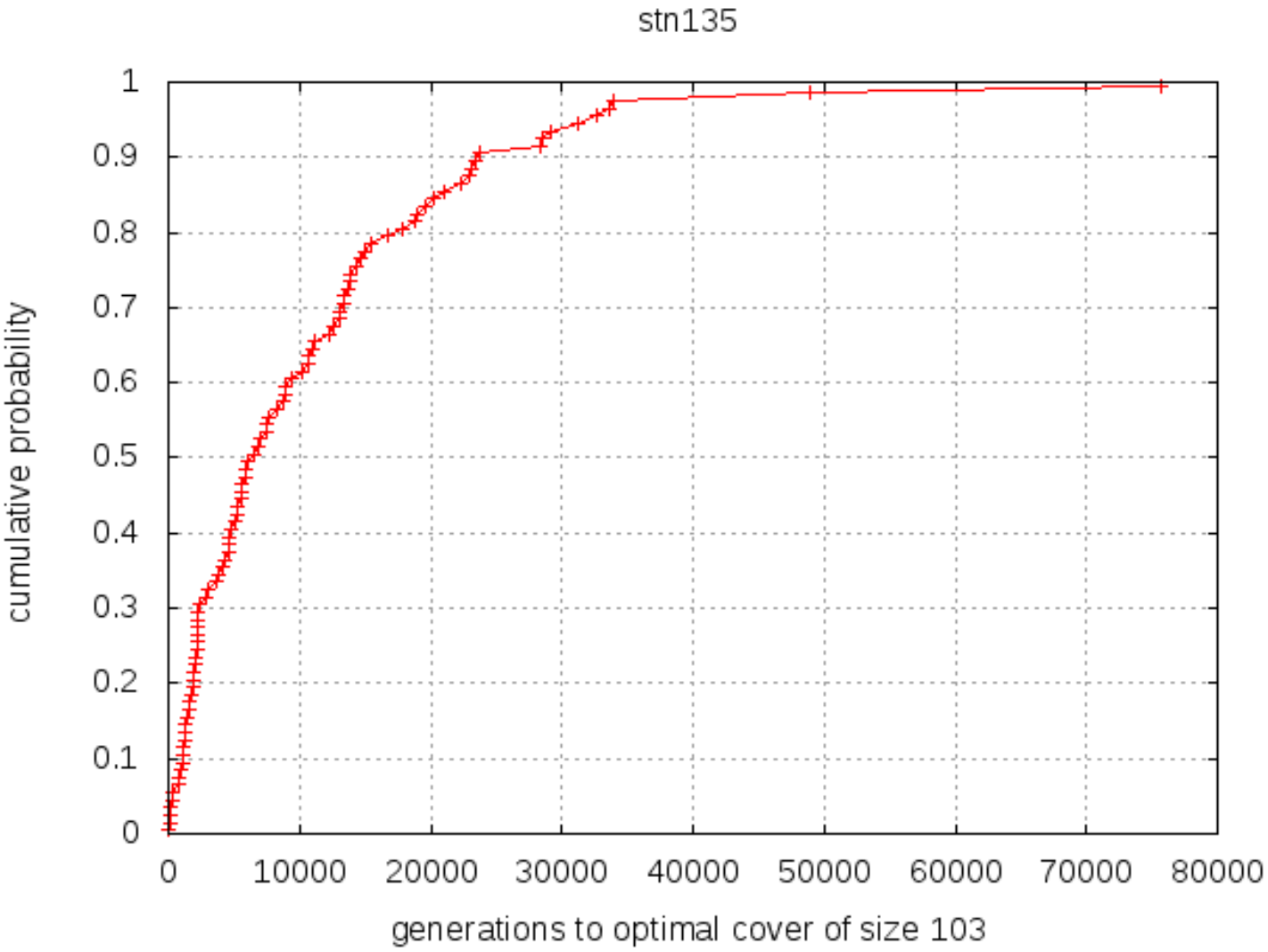
39 of the 100 runs required over 10,000 iterations



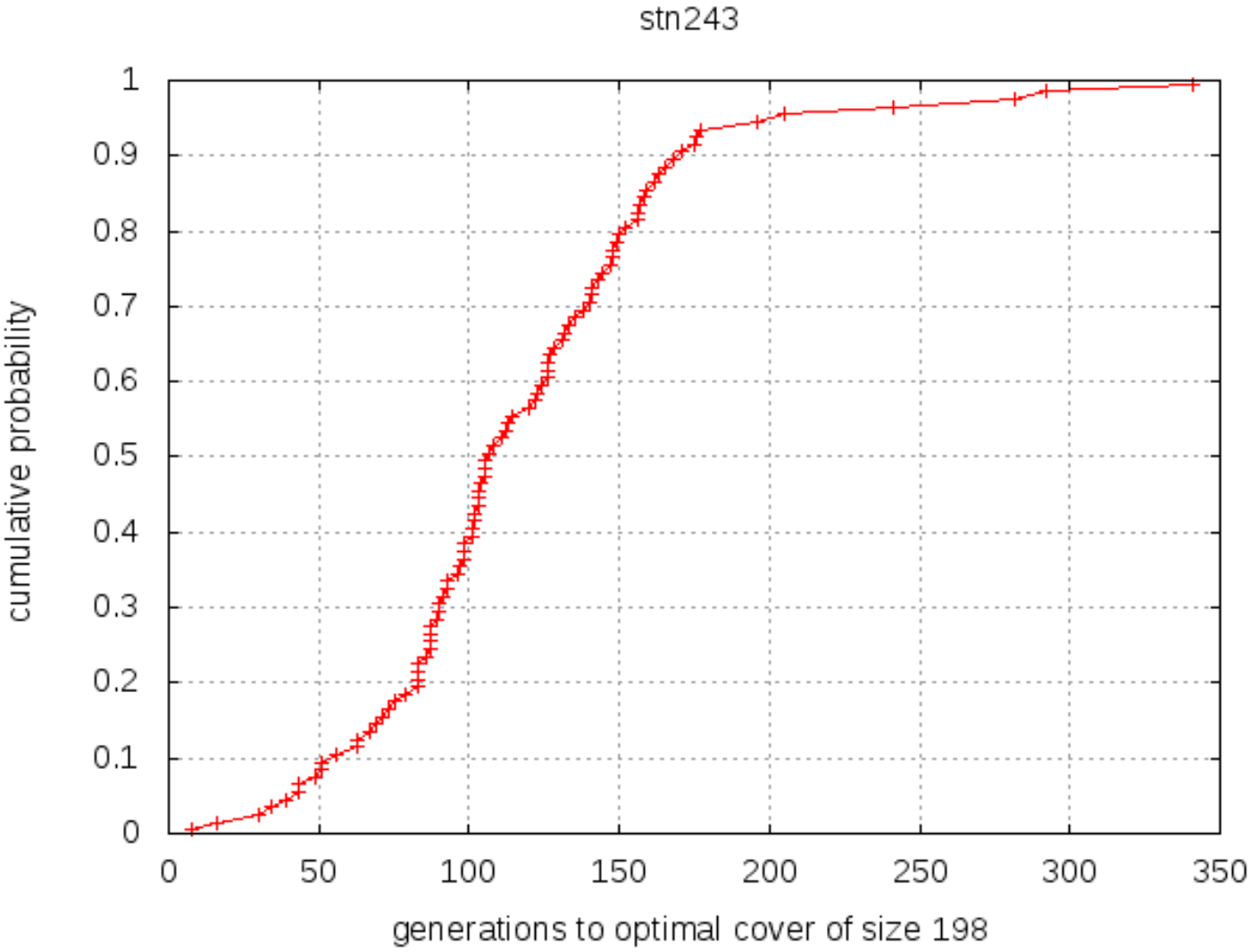
No run required fewer than 23 iterations



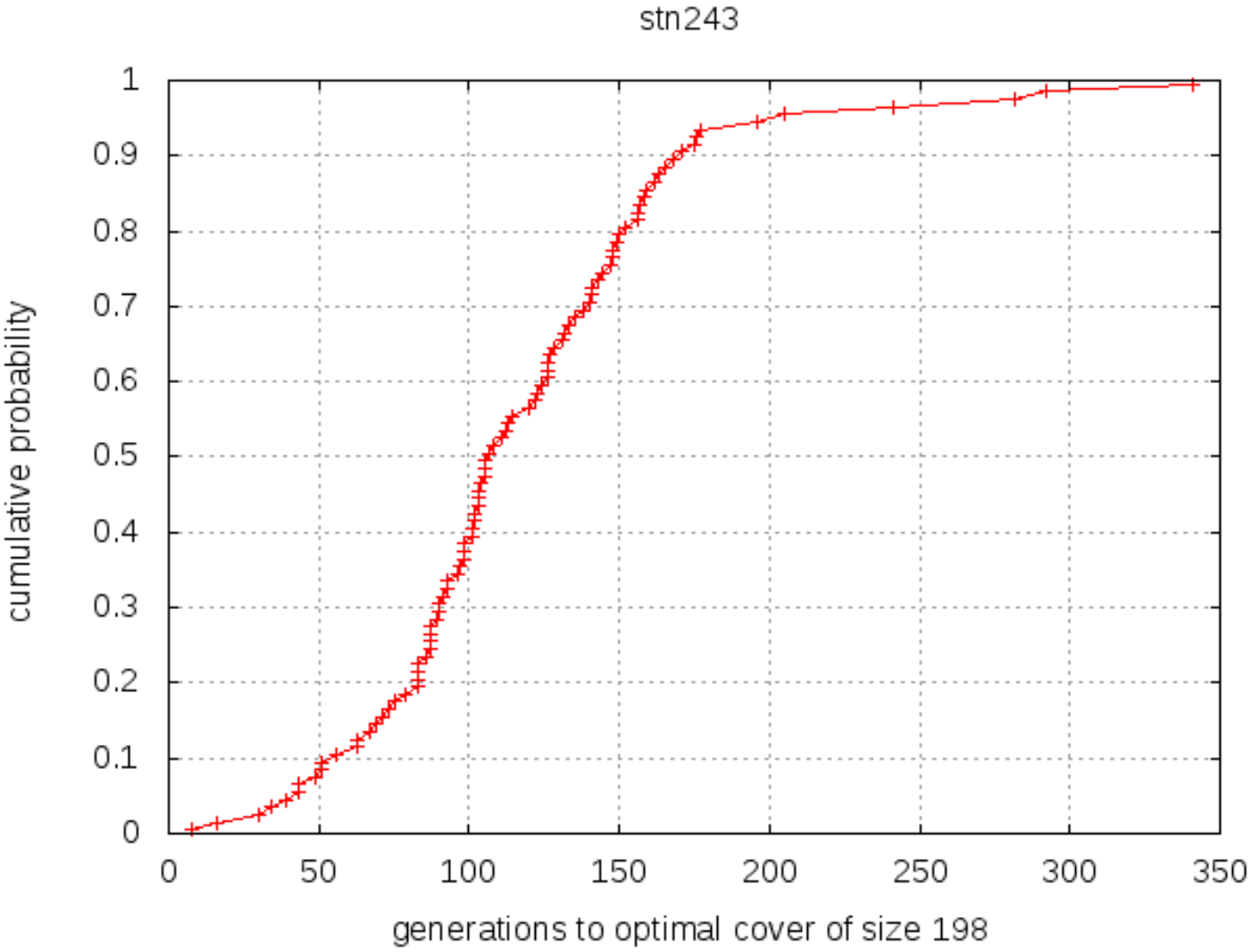
Longest run took 75,741 iterations



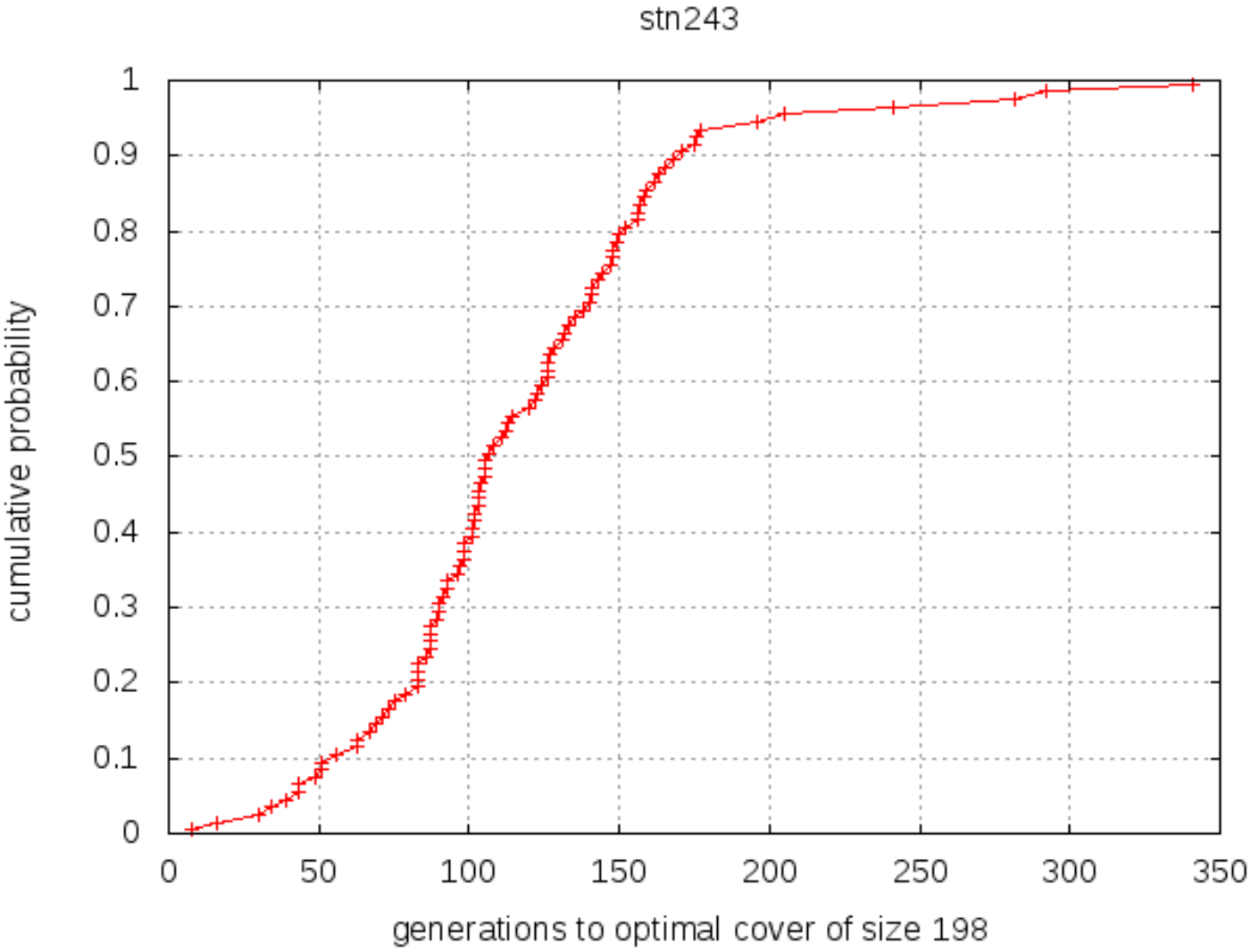
Time per 1000 generations: 19.91s (real), 316.70s (user), 0.85s (sys)



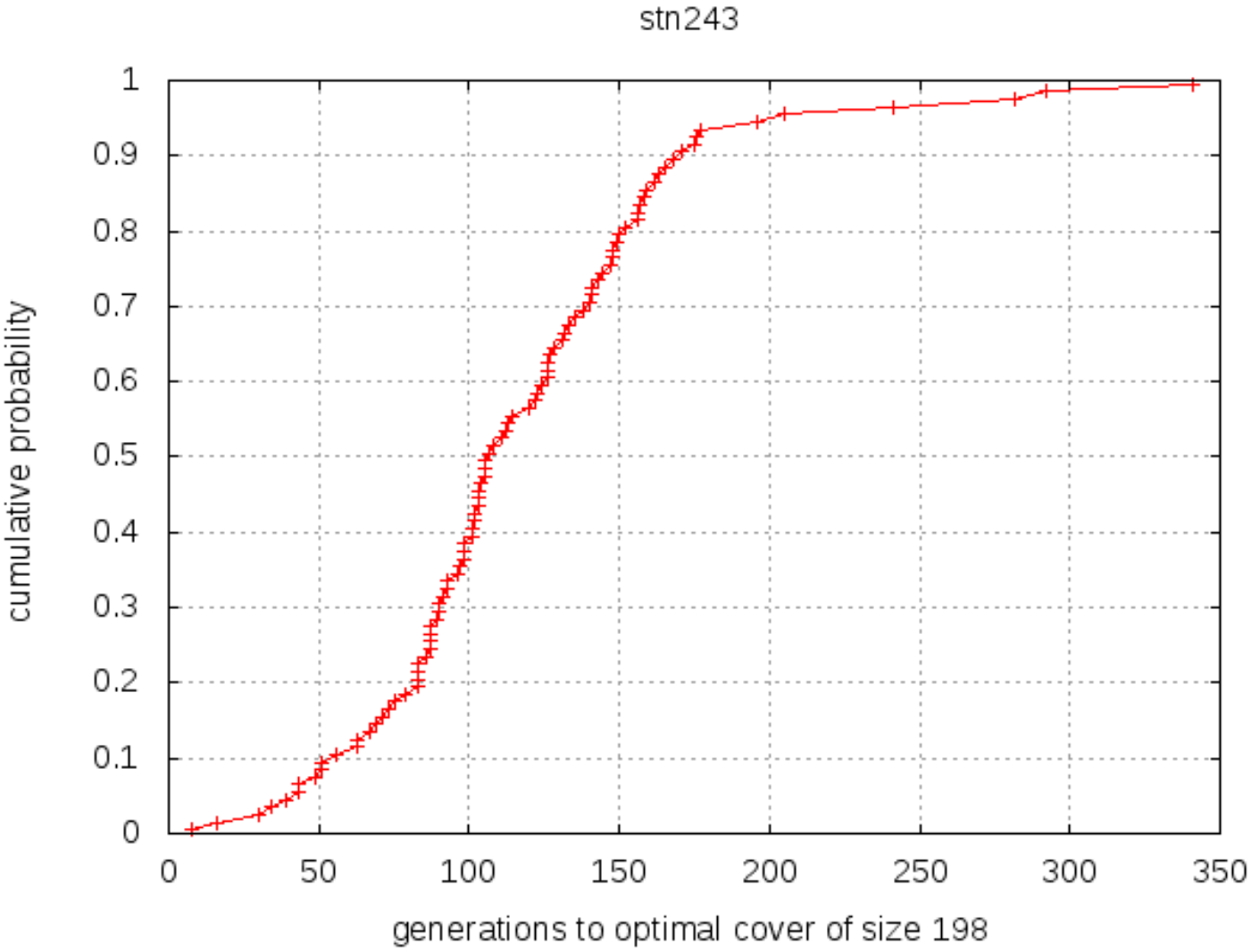
Instance stn243



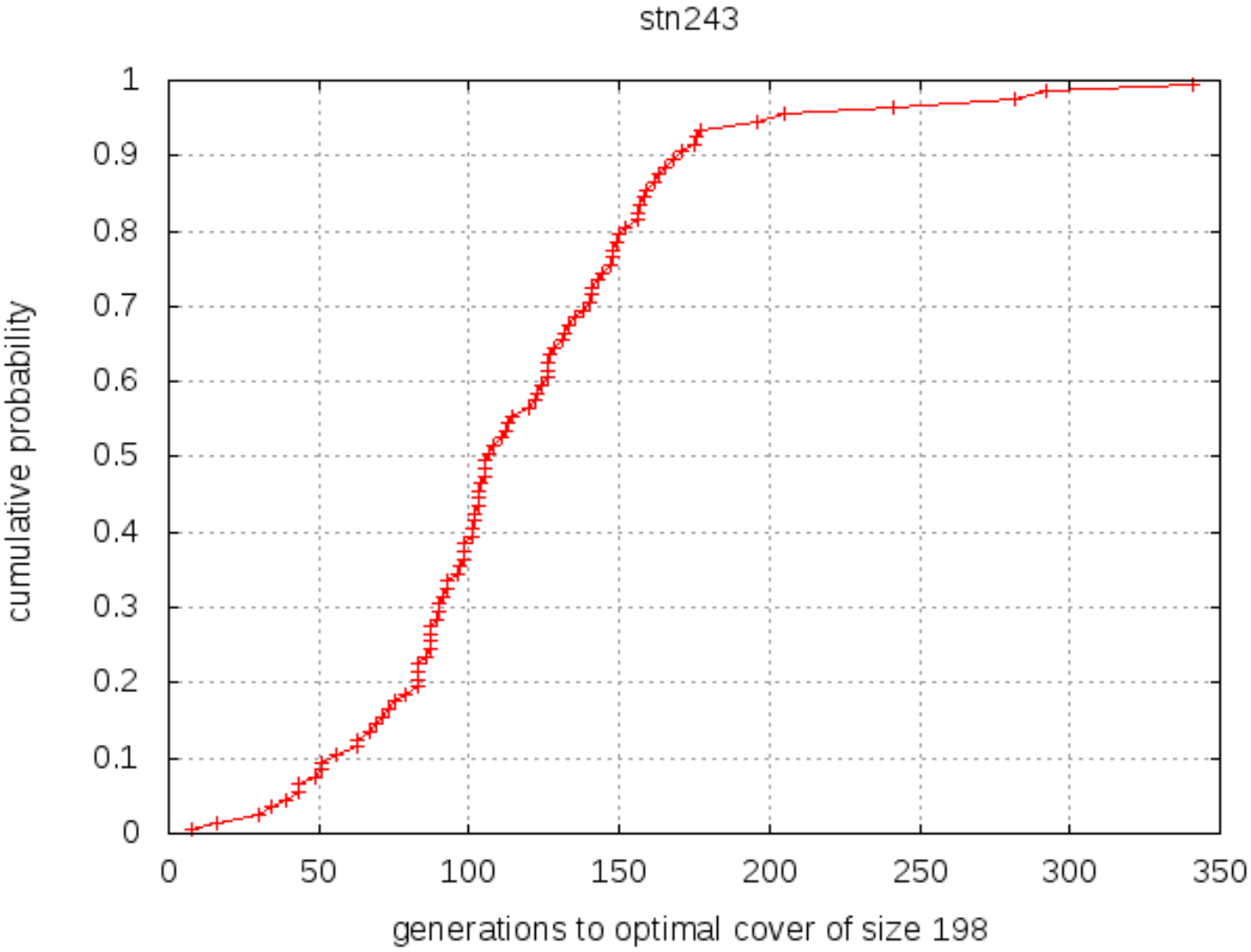
Appears to be much easier than stn135



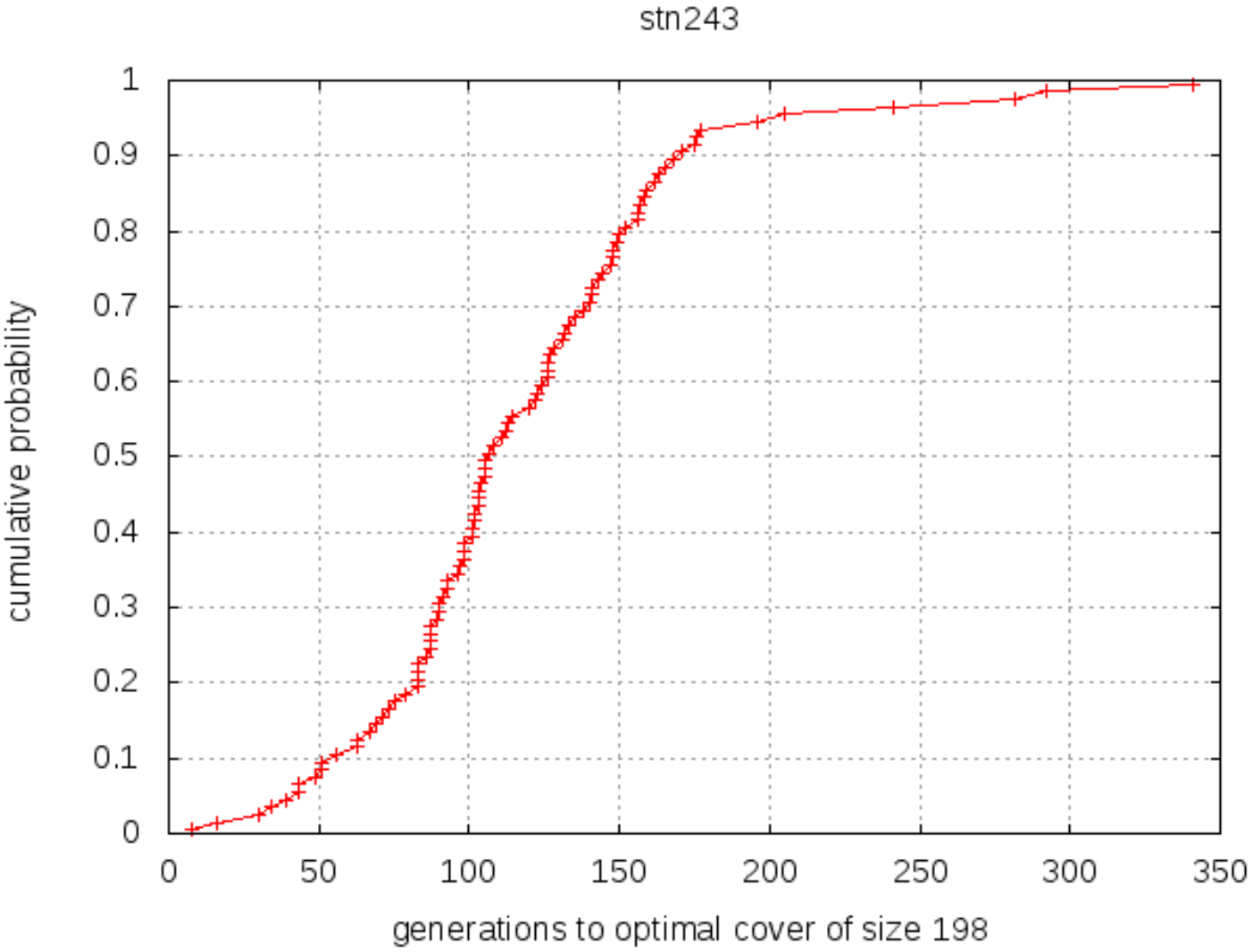
39/100 runs required fewer than 100 generations



95/100 runs required fewer than 200 generations



The longest of the 100 runs took 341 generations



Time per 1000 generations: 68.60s (real), 1095.19s (user), 0.79s (sys)

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

We conducted 100 independent runs simulating a random multi-start algorithm.

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

We conducted 100 independent runs simulating a random multi-start algorithm.

Each run consisted of 1000 generations with three populations, each with an elite set of size 1 and a mutant set of size 999.

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

We conducted 100 independent runs simulating a random multi-start algorithm.

Each run consisted of 1000 generations with three populations, each with an elite set of size 1 and a mutant set of size 999.

At each iteration 2997 random solutions are generated, each evaluated with the decoder.

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

We conducted 100 independent runs simulating a random multi-start algorithm.

Each run consisted of 1000 generations with three populations, each with an elite set of size 1 and a mutant set of size 999.

At each iteration 2997 random solutions are generated, each evaluated with the decoder.

Mating never takes place since elite and mutants make up the entire population.

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

About 300 million solutions were generated.

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

About 300 million solutions were generated.

The random multi-start was far from finding an optimal cover of size 198.

Simulation of random multi-start on stn243

To show that success of BRKGA to consistently find covers of size 198 on stn243 was not due to the decoder alone ...

About 300 million solutions were generated.

The random multi-start was far from finding an optimal cover of size 198.

It found covers of size 202 in 9/100 runs and of size 203 in the remaining 91/100.

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

For stn405 ... three runs found covers of size 335.

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

For stn405 ... three runs found covers of size 335.

Run 1 found the cover after 203 generations.

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

For stn405 ... three runs found covers of size 335.

Run 1 found the cover after 203 generations.

Run 2 ... after 5165 generations.

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

For stn405 ... three runs found covers of size 335.

Run 1 found the cover after 203 generations.

Run 2 ... after 5165 generations.

Run 3 ... after 2074 generations.

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

For stn405 ... three runs found covers of size 335.

Run 1 found the cover after 203 generations.

Run 2 ... after 5165 generations.

Run 3 ... after 2074 generations.

Time per 1000 generations: 796.82s (real), 12723.40s (user), 11.67s (sys)

Solution 1

1	2	3	4	5	31	32
33	34	35	56	57	58	59
60	86	87	88	89	90	91
92	93	94	95	106	107	108
109	110	146	147	148	149	150
171	172	173	174	175	201	202
203	204	205	221	222	223	224
225	226	227	228	229	230	266
267	268	269	270	271	272	273
274	275	306	307	308	309	310

Solution 2

6	7	8	9	10	26	27
28	29	30	41	42	43	44
45	51	52	53	54	55	71
72	73	74	75	86	87	88
89	90	151	152	153	154	155
196	197	198	199	200	226	227
228	229	230	261	262	263	264
265	286	287	288	289	290	331
332	333	334	335	361	362	363
364	365	396	397	398	399	400

Solution 3

6	7	8	9	10	21	22
23	24	25	31	32	33	34
35	46	47	48	49	50	76
77	78	79	80	96	97	98
99	100	136	137	138	139	140
151	152	153	154	155	176	177
178	179	180	196	197	198	199
200	251	252	253	254	255	266
267	268	269	270	341	342	343
344	345	371	372	373	374	375

Indices of 405 − 335 = 70 zeroes of covers of size 335 for stn405

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

For stn729 ... one run found a cover of size 617 after 1601 generations.

Experiments on the two largest instances

For instances stn405 and stn729: ran GA and stopped after 5000 generations without improvement.

For both instances, GA found improved solutions ...

For stn729 ... one run found a cover of size 617 after 1601 generations.

Time per 1000 generations: 6099.40s (real), 93946.68s (user), 498.00s (sys)

3	5	11	12	27	36	39	43
52	54	56	63	70	73	74	85
94	121	128	142	159	166	167	176
177	181	197	200	201	214	215	220
225	230	237	239	245	252	255	263
264	277	279	283	288	291	299	309
313	322	323	331	333	334	343	344
355	357	364	365	377	382	390	392
400	405	410	430	437	446	470	483
497	509	520	535	548	550	560	561
565	567	570	578	580	590	591	599
600	608	614	621	627	629	632	639
648	652	661	663	669	673	680	682
693	697	699	705	709	712	717	723

Indices of 729 – 617 = 112 zeroes of cover of size 617 for stn729

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel, including ...

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel, including ...

Generation of random key vectors in initial population and mutants at each generation with corresponding calls to random number generator;

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel, including ...

Generation of random key vectors in initial population and mutants at each generation with corresponding calls to random number generator;

Crossover at each generation to produce offspring;

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel, including ...

Generation of random key vectors in initial population and mutants at each generation with corresponding calls to random number generator;

Crossover at each generation to produce offspring;

Periodic exchange of elite solutions among multiple populations;

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel, including ...

Generation of random key vectors in initial population and mutants at each generation with corresponding calls to random number generator;

Crossover at each generation to produce offspring;

Periodic exchange of elite solutions among multiple populations;

Sorting of population by fitness values;

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel, including ...

Generation of random key vectors in initial population and mutants at each generation with corresponding calls to random number generator;

Crossover at each generation to produce offspring;

Periodic exchange of elite solutions among multiple populations;

Sorting of population by fitness values;

Copying elite solutions to next generation.

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel ...

Consequently 100% efficiency (linear speedup) cannot be expected.

Computing covers with a parallel implementation

The BRKGA does decoding in parallel. Decoding is the major computational bottleneck of the BRKGA.

There are other tasks that are not done in parallel ...

Consequently 100% efficiency (linear speedup) cannot be expected.

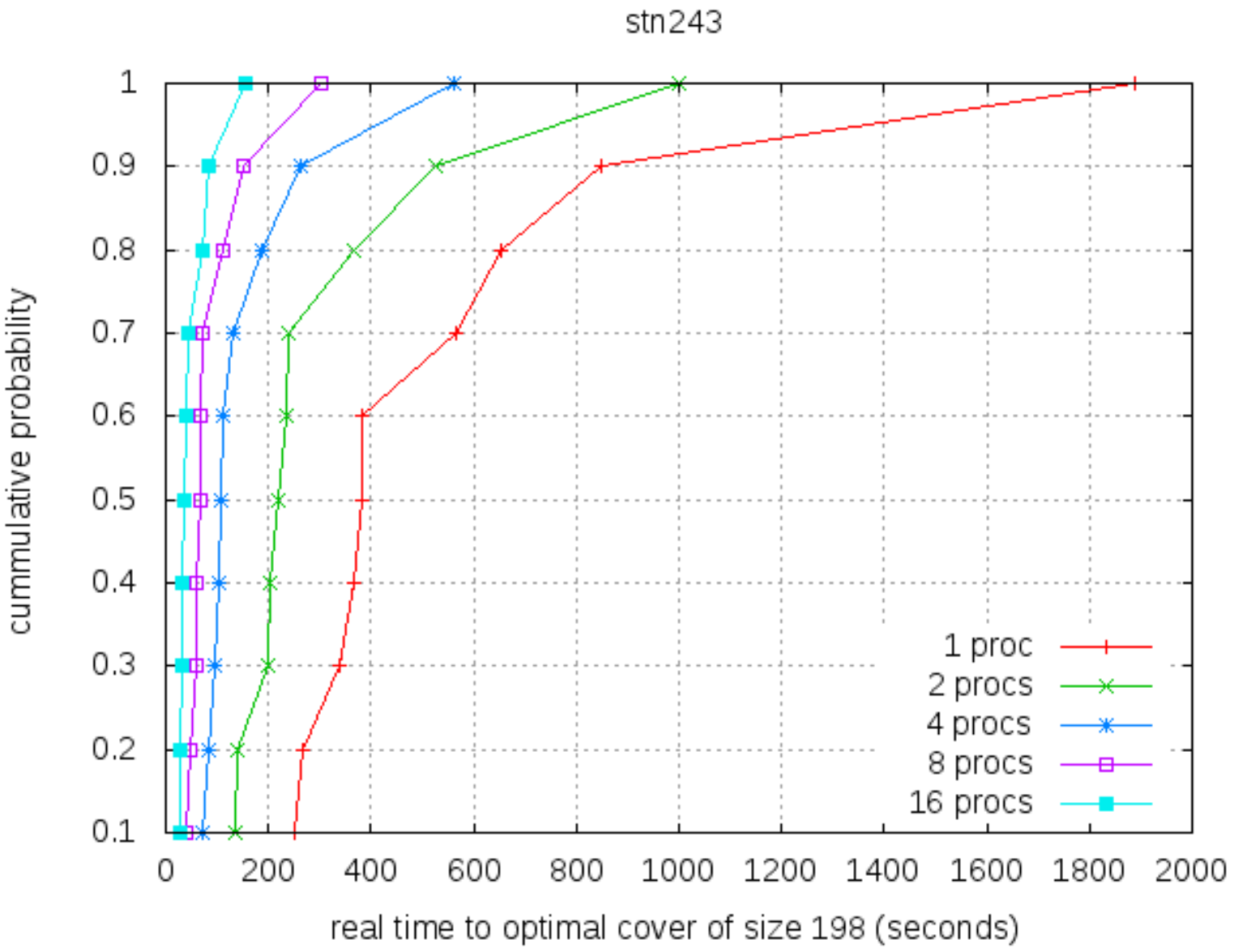
Nevertheless, we observe significant speedup.

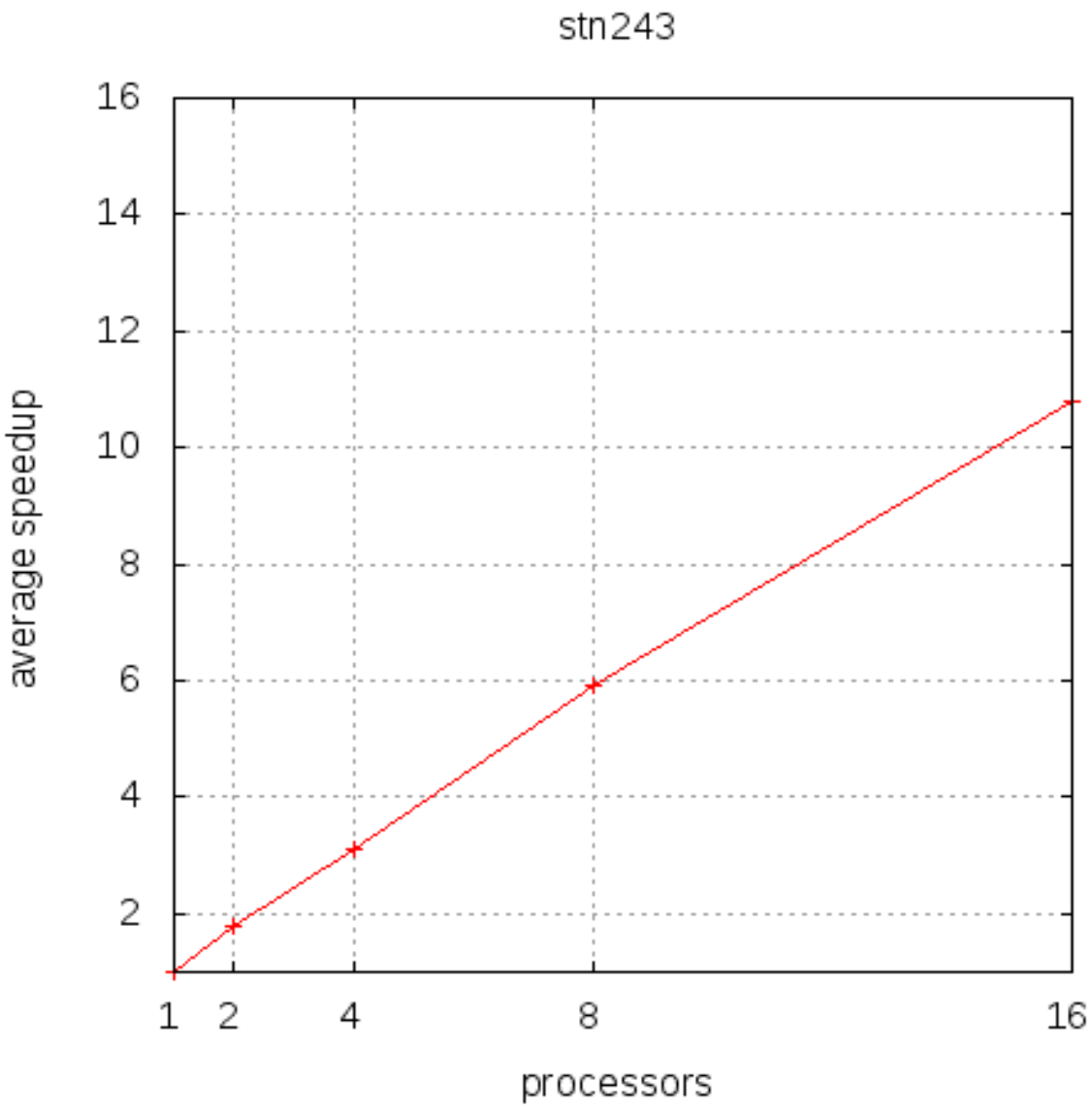
Computing covers with a parallel implementation

To illustrate the parallel efficiency of the BRKGA we carried out the following experiment on instance stn243 ...

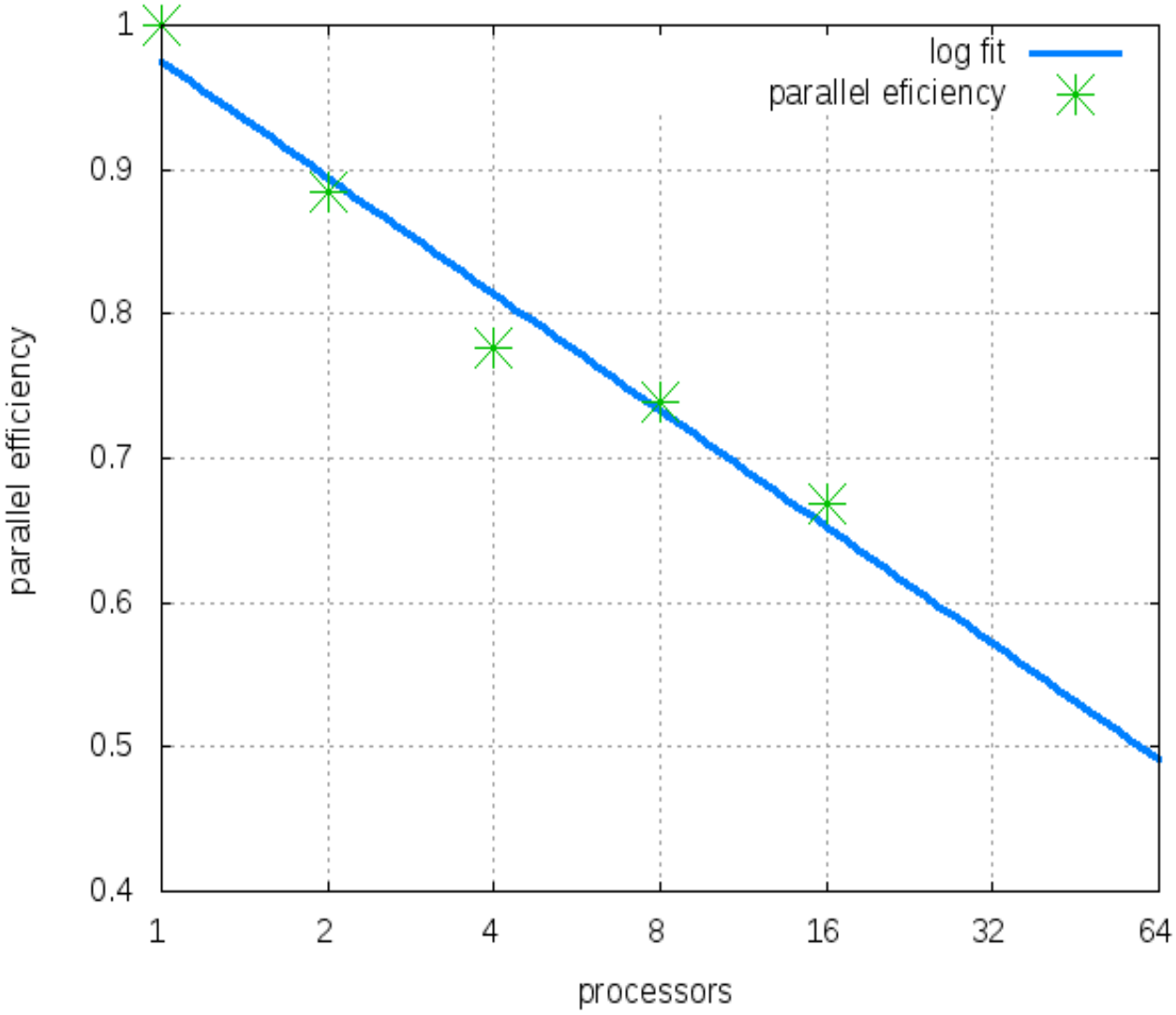
On each of five processor configurations (single processor, two, four, eight, and 16 processors) ...

We made 10 independent runs of the BRKGA, stopping when an optimal cover of size 198 was found.

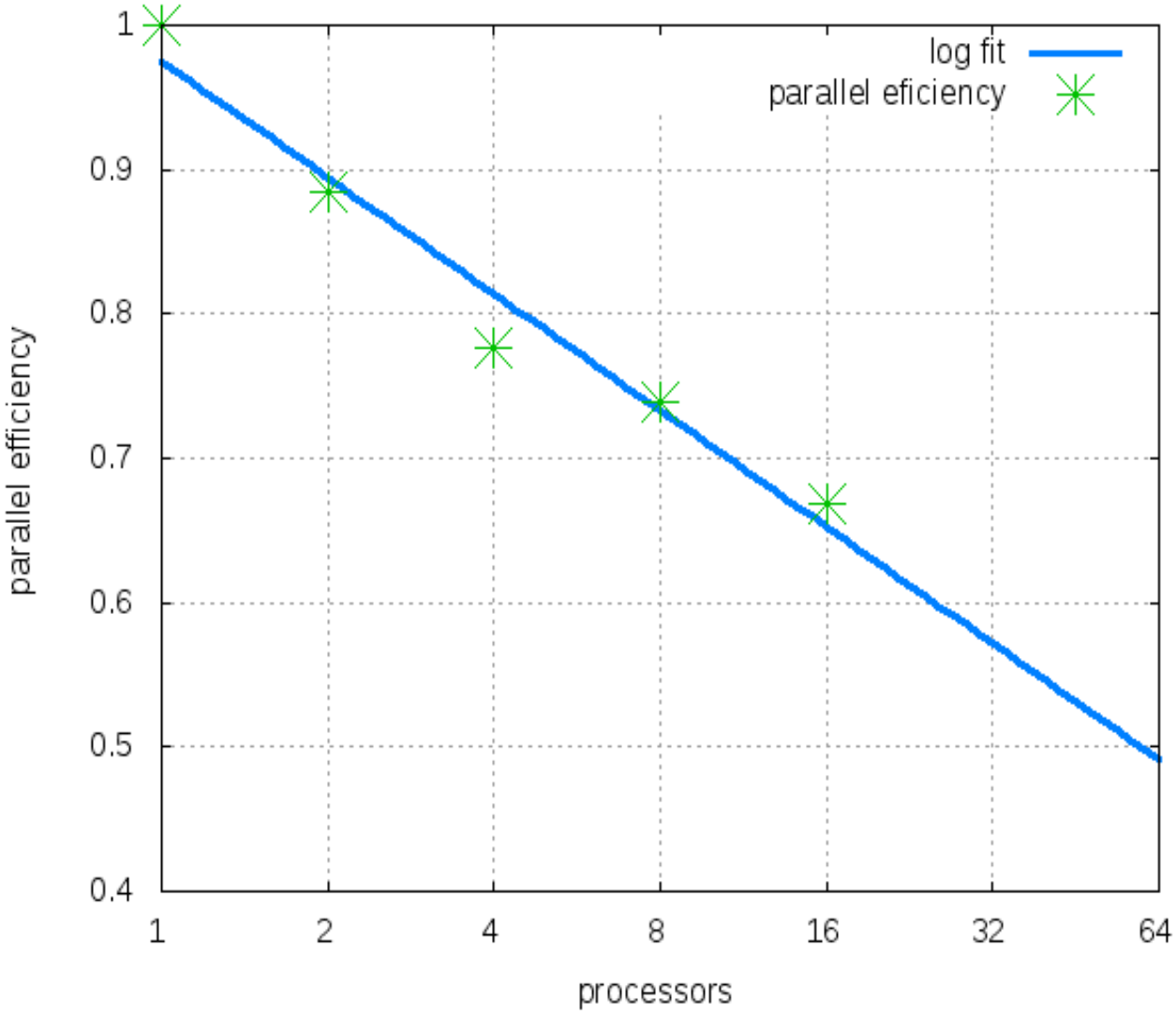




Speedup with 16 processors is almost 11-fold.



Parallel efficiency is $t_1 / [\, p - t_p \,]$, where p is the number of processors and t_k is the real time using k processors.



Log fit suggests that with 64 processors we can still expect a 32-fold speedup.

Concluding remarks

Concluding remarks

Introduced a biased random-key genetic algorithm for the Steiner triple covering problem.

Concluding remarks

Introduced a biased random-key genetic algorithm for the Steiner triple covering problem.

The parallel, multi-population, implementation of the BRKGA not only found optimal covers for all instances with known optimal solution ...

Concluding remarks

Introduced a biased random-key genetic algorithm for the Steiner triple covering problem.

The parallel, multi-population, implementation of the BRKGA not only found optimal covers for all instances with known optimal solution ...

It also found new best known covers for two recently introduced instances ... of size 335 for stn405 and 617 for stn729

Concluding remarks

Introduced a biased random-key genetic algorithm for the Steiner triple covering problem.

The parallel, multi-population, implementation of the BRKGA not only found optimal covers for all instances with known optimal solution ...

It also found new best known covers for two recently introduced instances ... of size 335 for stn405 and 617 for stn729

The parallel implementation achieved a speedup of 10.8 with 16 processors and is expected to achieve a speedup of about 32 with 64 processors

Concluding remarks

We have recently proposed BRKGAs for ...

Set covering by pairs;

Set covering with general costs;

Set k-covering with general costs.

The End

These slides and all of my papers cited in this talk can be downloaded from the homepage:

`http://www2.research.att.com/~mgcr`