Biased random-key genetic algorithms: A tutorial

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

mgcr@research.att.com

Tutorial given at CLAIO/SBPO 2012
Rio de Janeiro, Brazil ✤ September 2012

AT&T Shannon Laboratory
Florham Park, New Jersey

Summary: Day 1

- Basic concepts of combinatorial and continuous global optimization
- Basic concepts of genetic algorithms
- Random-key genetic algorithm of Bean (1994)
- Biased random-key genetic algorithms (BRKGA)
  - Encoding / Decoding
  - Initial population
  - Evolutionary mechanisms
  - Problem independent / problem dependent components
  - Multi-start strategy
  - Restart strategy
  - Multi-population strategy
  - Specifying a BRKGA
- Application programming interface (API) for BRKGA

Summary: Day 2

- Applications of BRKGA
  - Set covering
  - Packing rectangles
  - Packet routing on the Internet
  - Handover minimization in mobility networks
  - Continuous global optimization
- Overview of literature & concluding remarks

Combinatorial and Continuous Global Optimization

Combinatorial Optimization

Combinatorial optimization: process of finding the best, or optimal, solution for problems with a discrete set of feasible solutions.

Applications: routing, scheduling, packing, inventory and production management, location, logic, and assignment of resources, among many others.

Economic impact: transportation (airlines, trucking, rail, and shipping), forestry, manufacturing, logistics, aerospace, energy (electrical power, petroleum, and natural gas), agriculture, biotechnology, financial services, and telecommunications, among many others.

Combinatorial Optimization

Given:

discrete set of feasible solutions X

objective function $f(x): x \in X \rightarrow R$

Objective (minimization):

find $x \in X : f(x) \leq f(y), \forall\, y \in X$

Combinatorial Optimization

Much progress in recent years on finding exact (provably optimal) solutions: dynamic programming, cutting planes, branch and cut, …

Many hard combinatorial optimization problems are still not solved exactly and require good solution methods.

Combinatorial Optimization

Approximation algorithms are guaranteed to find in polynomial-time a solution within a given factor of the optimal.

Combinatorial Optimization

Approximation algorithms are guaranteed to find in polynomial-time a solution within a given factor of the optimal.

Sometimes the factor is too big, i.e. guaranteed solutions may be far from optimal

Some optimization problems (e.g. max clique, covering by pairs) cannot have approximation schemes unless P=NP

Combinatorial Optimization

Aim of heuristic methods for combinatorial optimization is to quickly produce good-quality solutions, without necessarily providing any guarantee of solution quality.

Continuous Global Optimization

Given:

continuous set of feasible solutions X

objective function $f(x): x \in X \rightarrow R$

Objective (minimization):

find $x \in X : f(x) \leq f(y), \forall\, y \in X$

Continuous Global Optimization

Given:

continuous set of feasible solutions X

objective function $f(x): x \in X \rightarrow R$

Objective (minimization):

find $x \in X : f(x) \leq f(y), \forall\, y \in X$

f(x) can be well-behaved or not, e.g. it can be non-convex, discontinuous, non-differentiable, a black-box, etc.

Continuous Box-Constrained Global Optimization

Here, the continuous set of solutions

$$X = [l_1,u_1] \times [l_2,u_2] \times \cdots \times [l_n,u_n]$$

is a hyper-rectangle, i.e. variables have lower and upper bounds.

Metaheuristics

Metaheuristics are heuristics to devise heuristics.

Metaheuristics

Metaheuristics are high level procedures that coordinate simple heuristics, such as local search, to find solutions that are of better quality than those found by the simple heuristics alone.

Metaheuristics

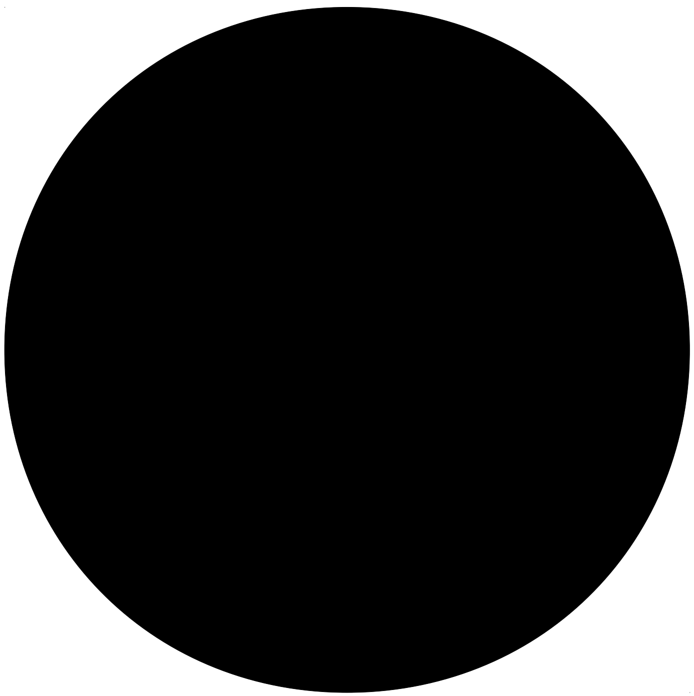
Metaheuristics are high level procedures that coordinate simple heuristics, such as local search, to find solutions that are of better quality than those found by the simple heuristics alone.

Examples: GRASP and C-GRASP, simulated annealing, genetic algorithms, tabu search, scatter search, ant colony optimization, variable neighborhood search, and biased random-key genetic algorithms (BRKGA).

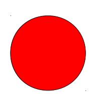
Genetic algorithms

Genetic algorithms

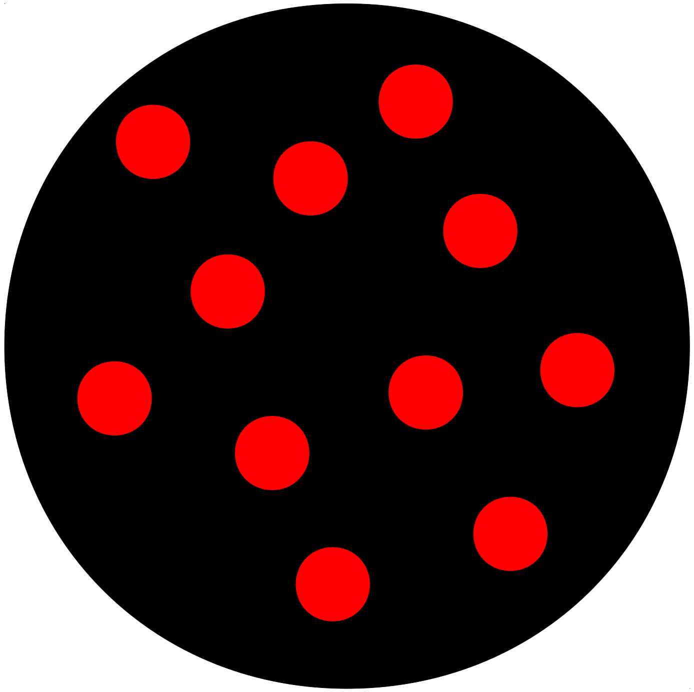
Holland (1975)



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

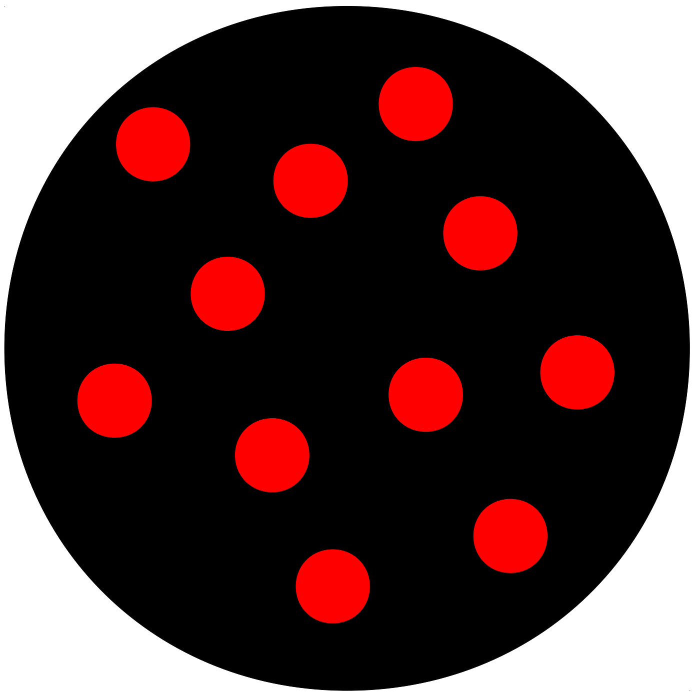
Individual: solution 

Genetic algorithms



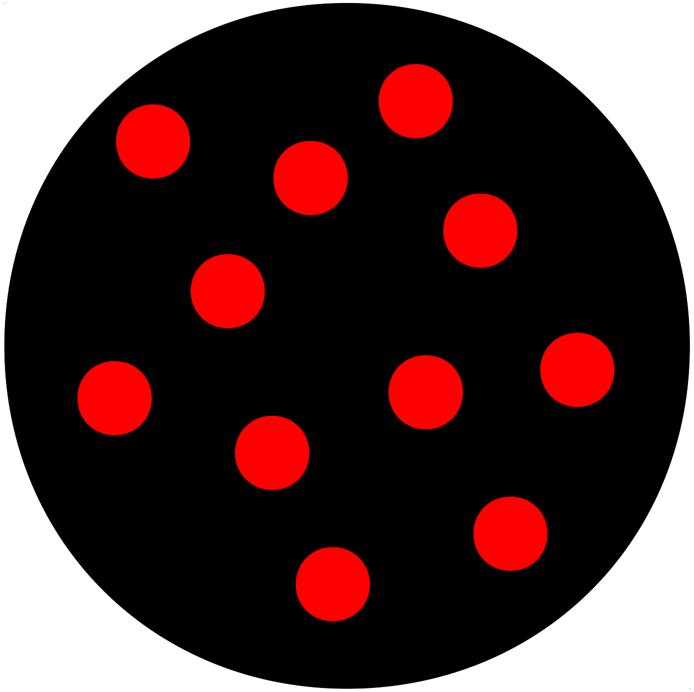
Individual: solution (chromosome = string of genes)
Population: set of fixed number of individuals

Genetic algorithms



Genetic algorithms evolve population applying Darwin's principle of survival of the fittest.

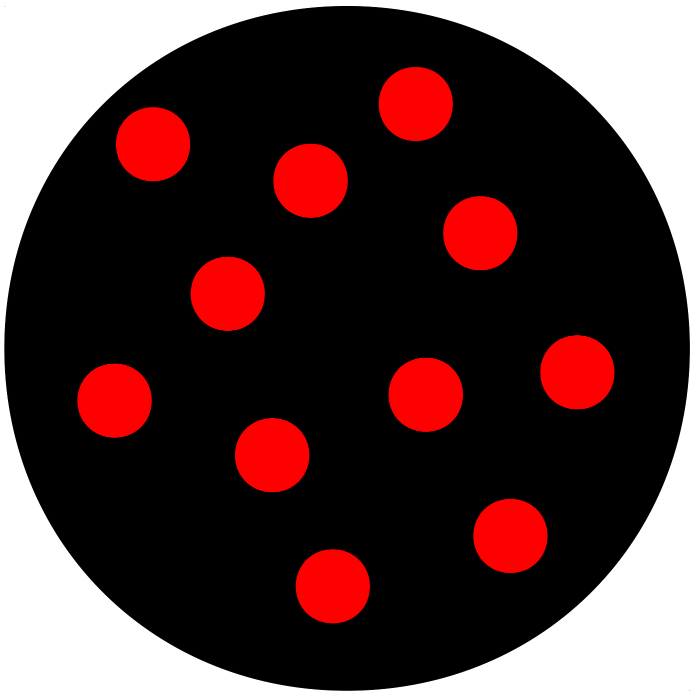
Genetic algorithms



Genetic algorithms evolve population applying Darwin's principle of survival of the fittest.

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

Genetic algorithms

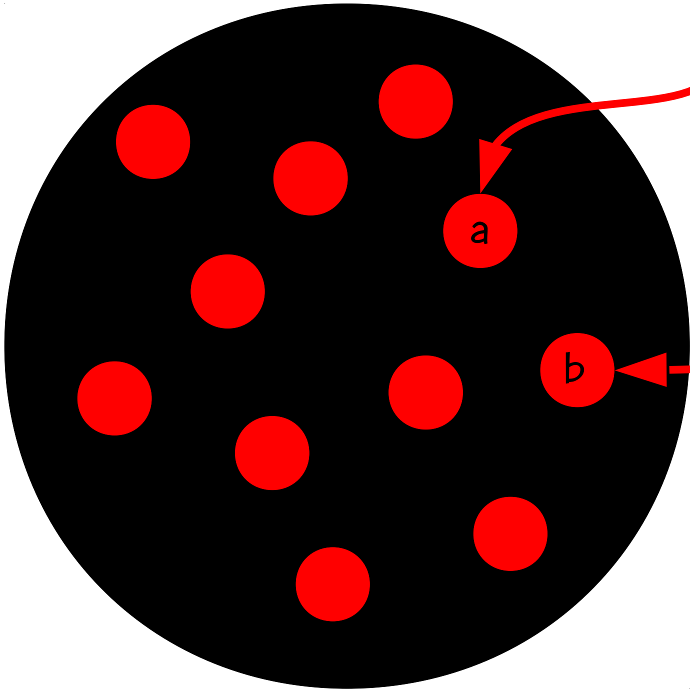


Genetic algorithms evolve population applying Darwin's principle of survival of the fittest.

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

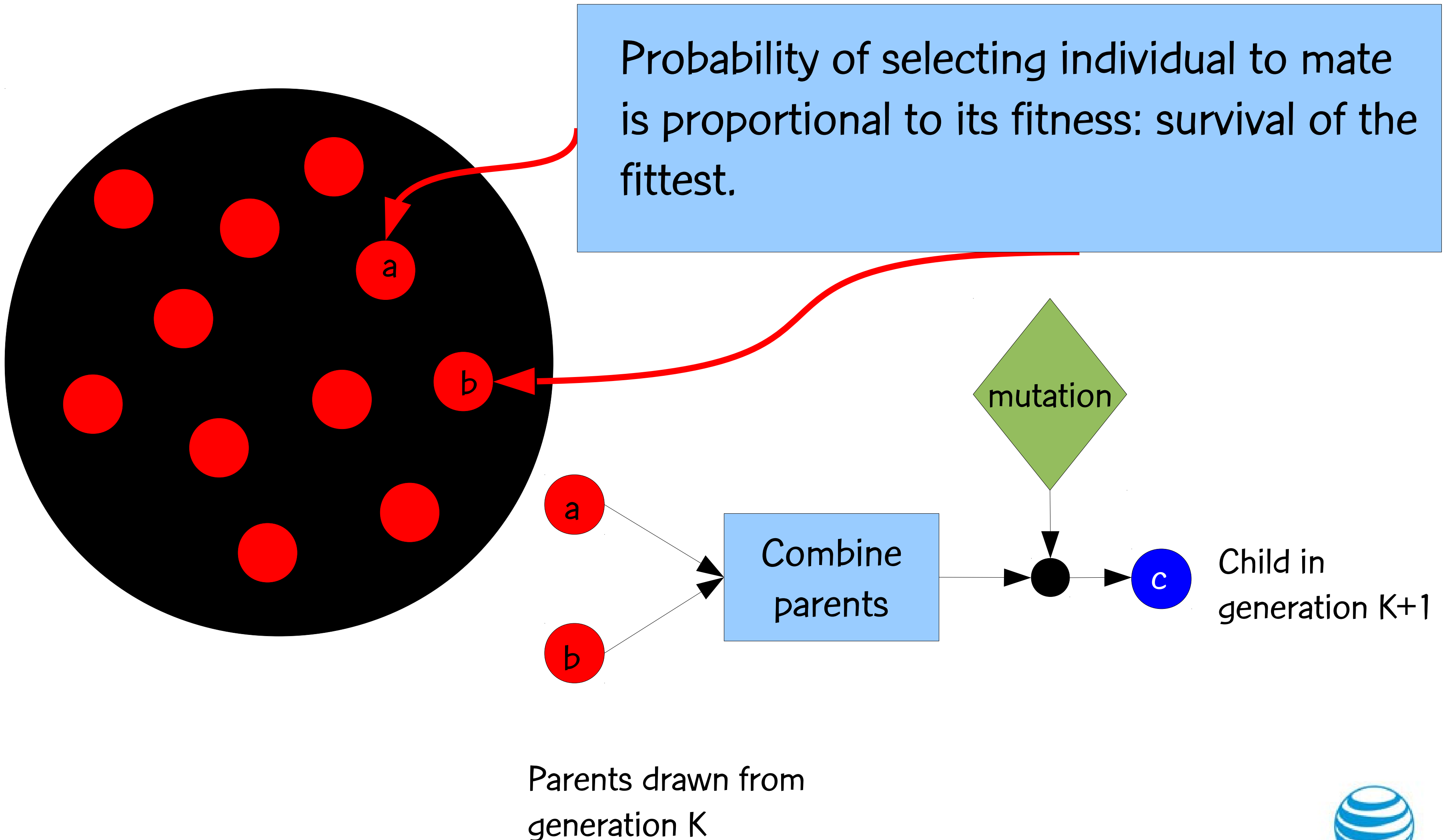
Individuals from one generation are combined to produce offspring that make up next generation.

Genetic algorithms

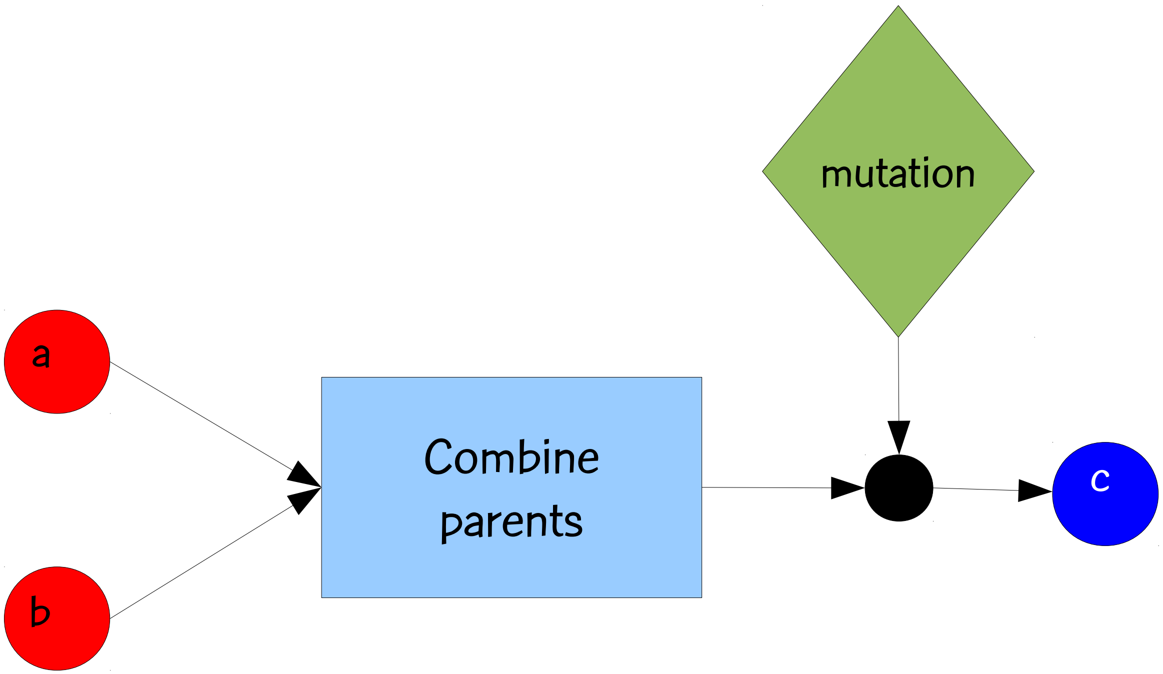


Probability of selecting individual to mate is proportional to its fitness: survival of the fittest.

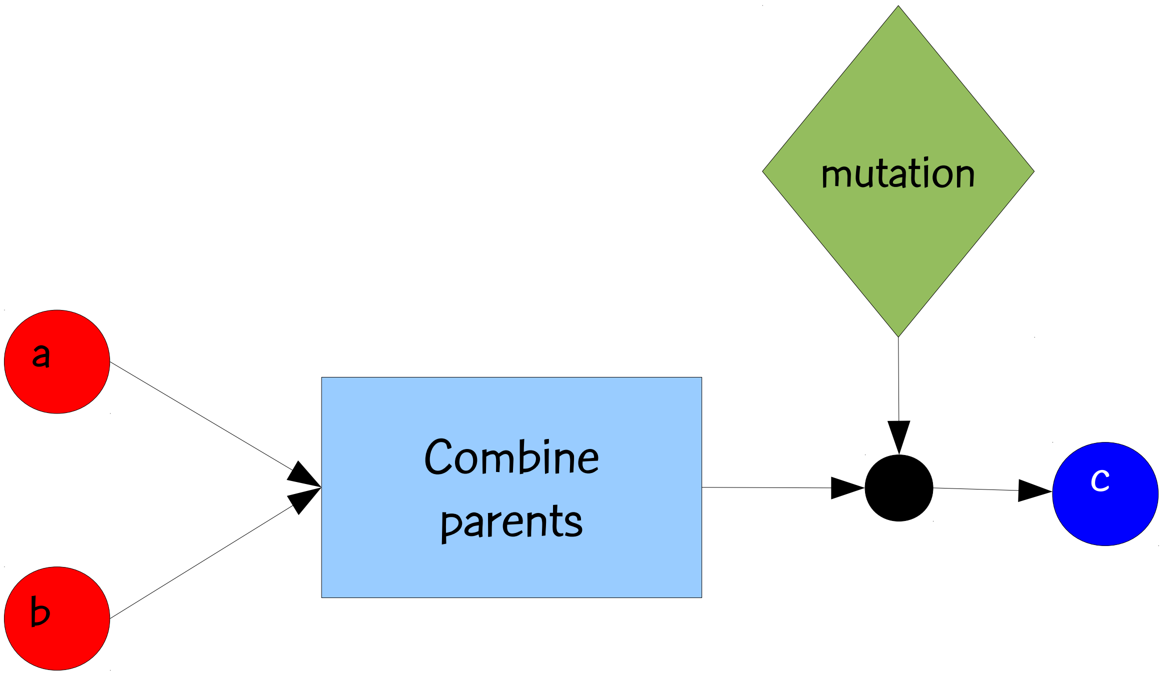
Genetic algorithms



Crossover and mutation



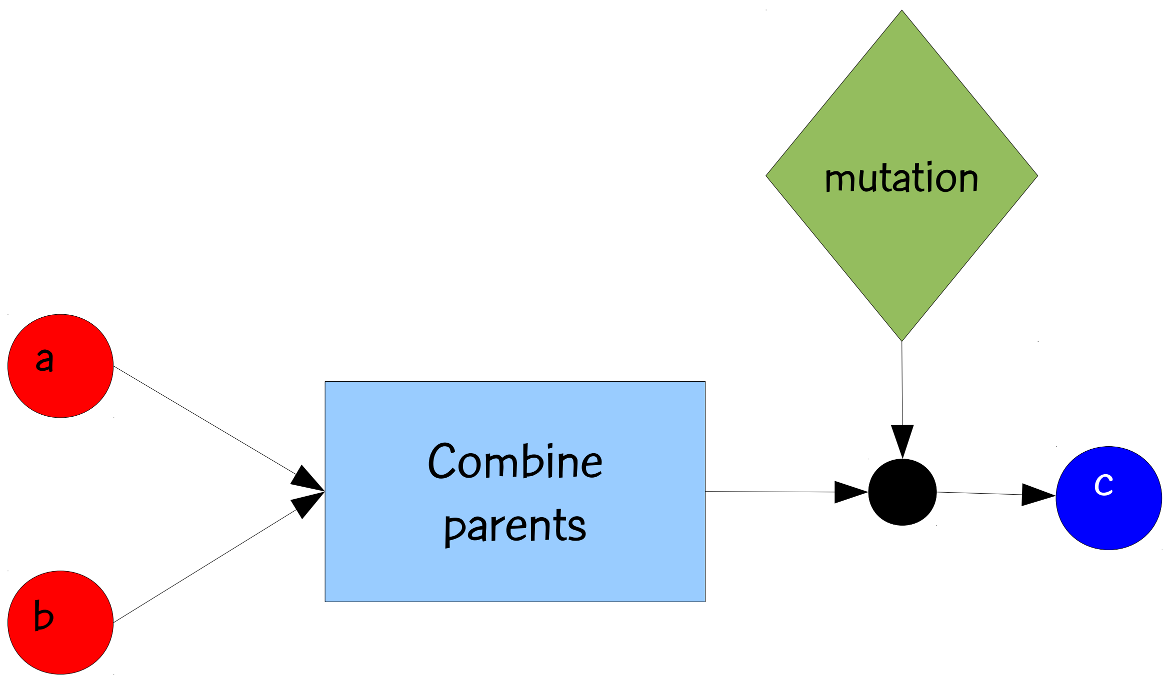
Crossover and mutation



Crossover: Combines parents ... passing along to offspring characteristics of each parent ...

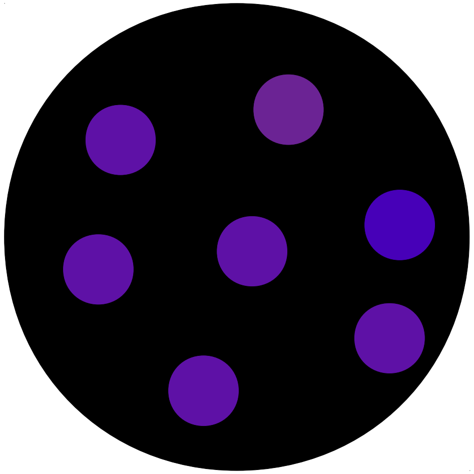
Intensification of search

Crossover and mutation

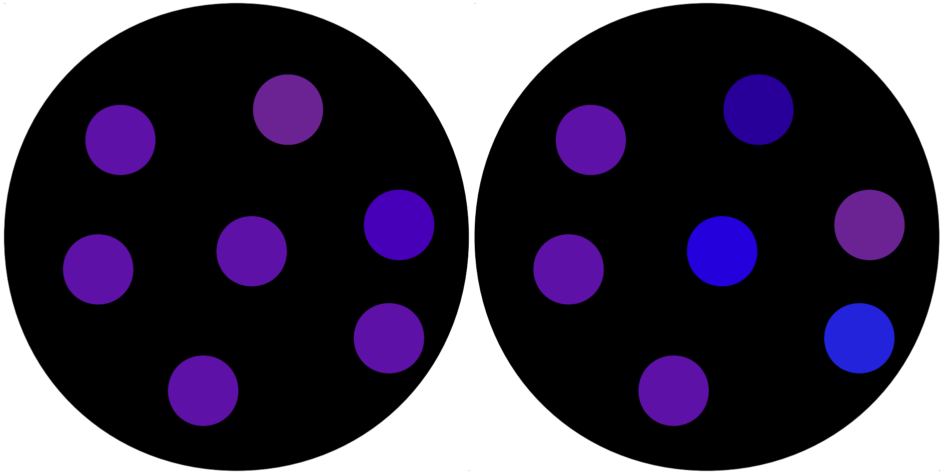


Mutation: Randomly changes chromosome of offspring ...
Driver of evolutionary process ...
Diversification of search

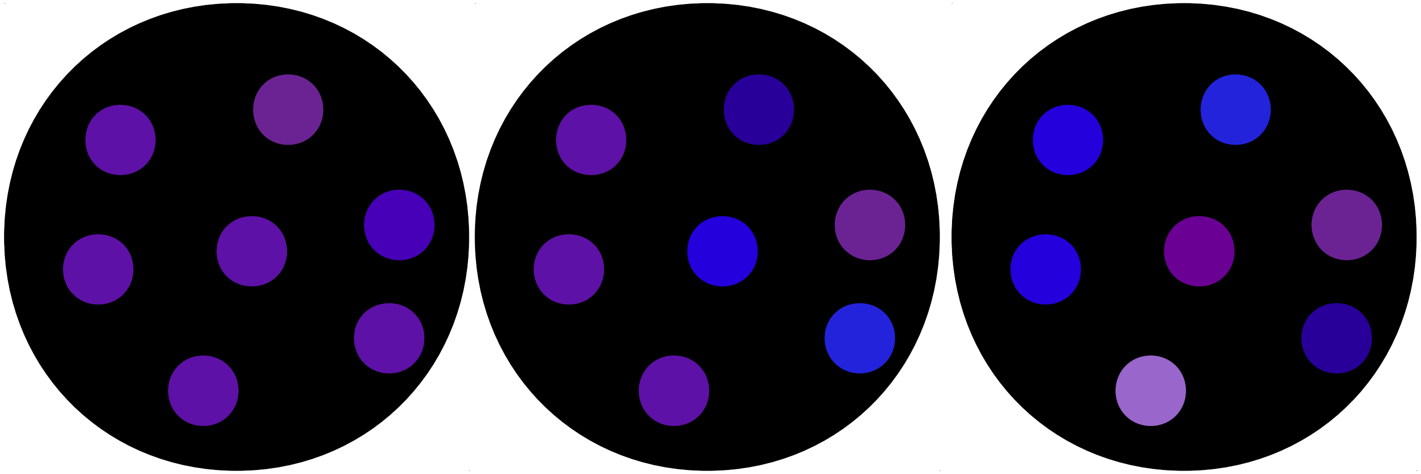
Evolution of solutions



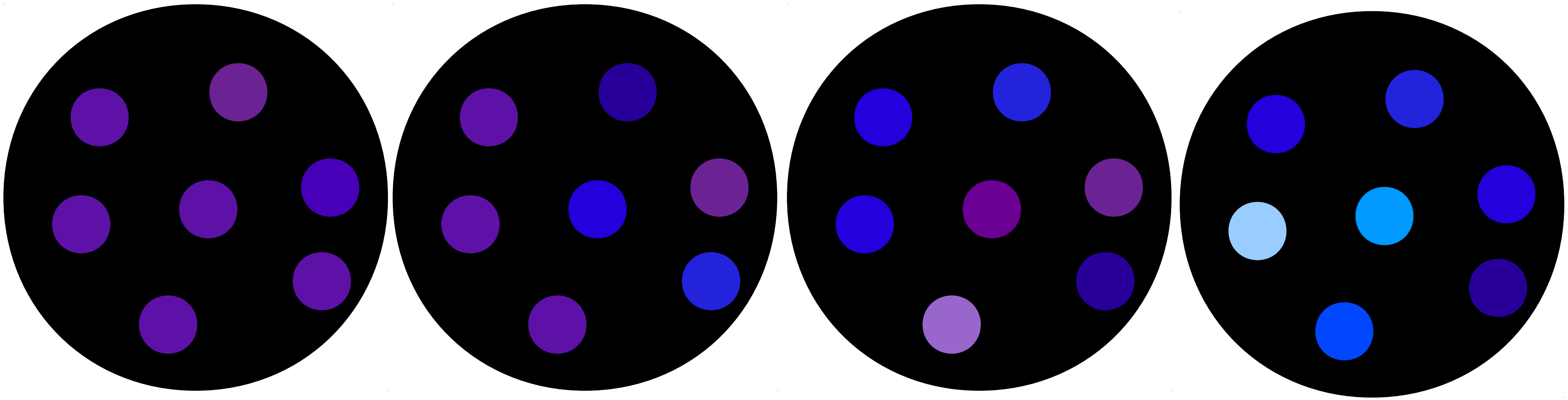
Evolution of solutions



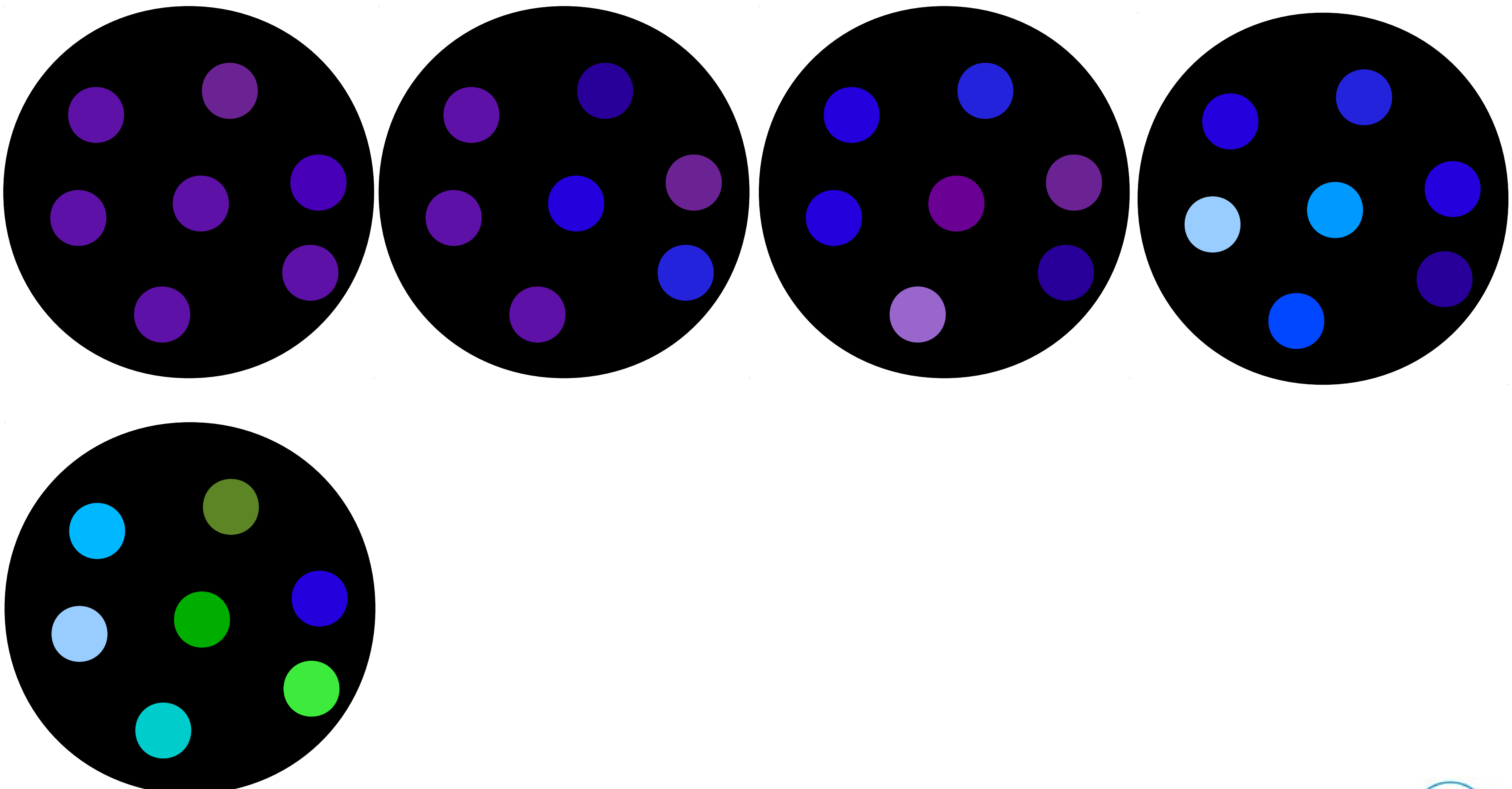
Evolution of solutions



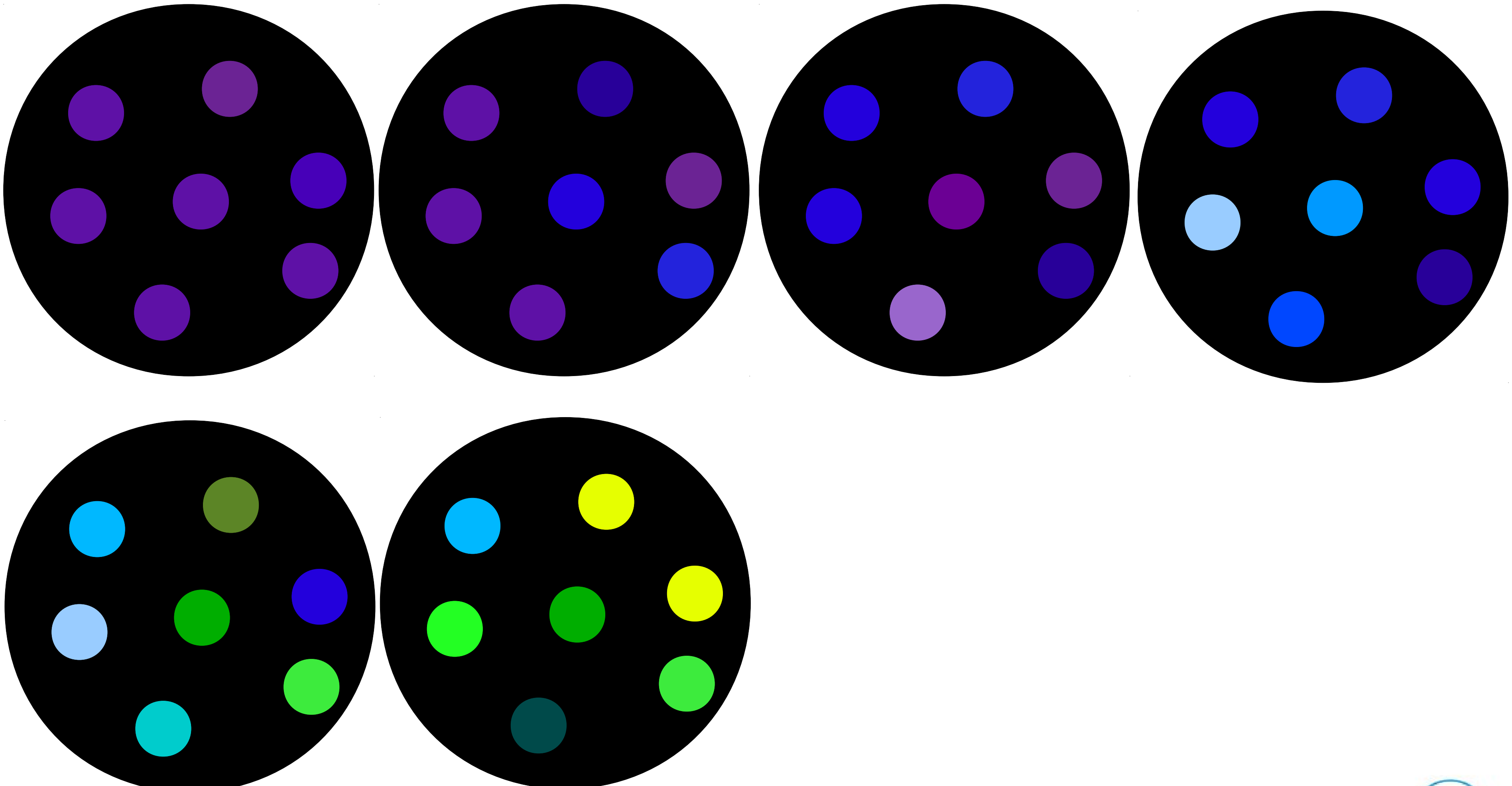
Evolution of solutions



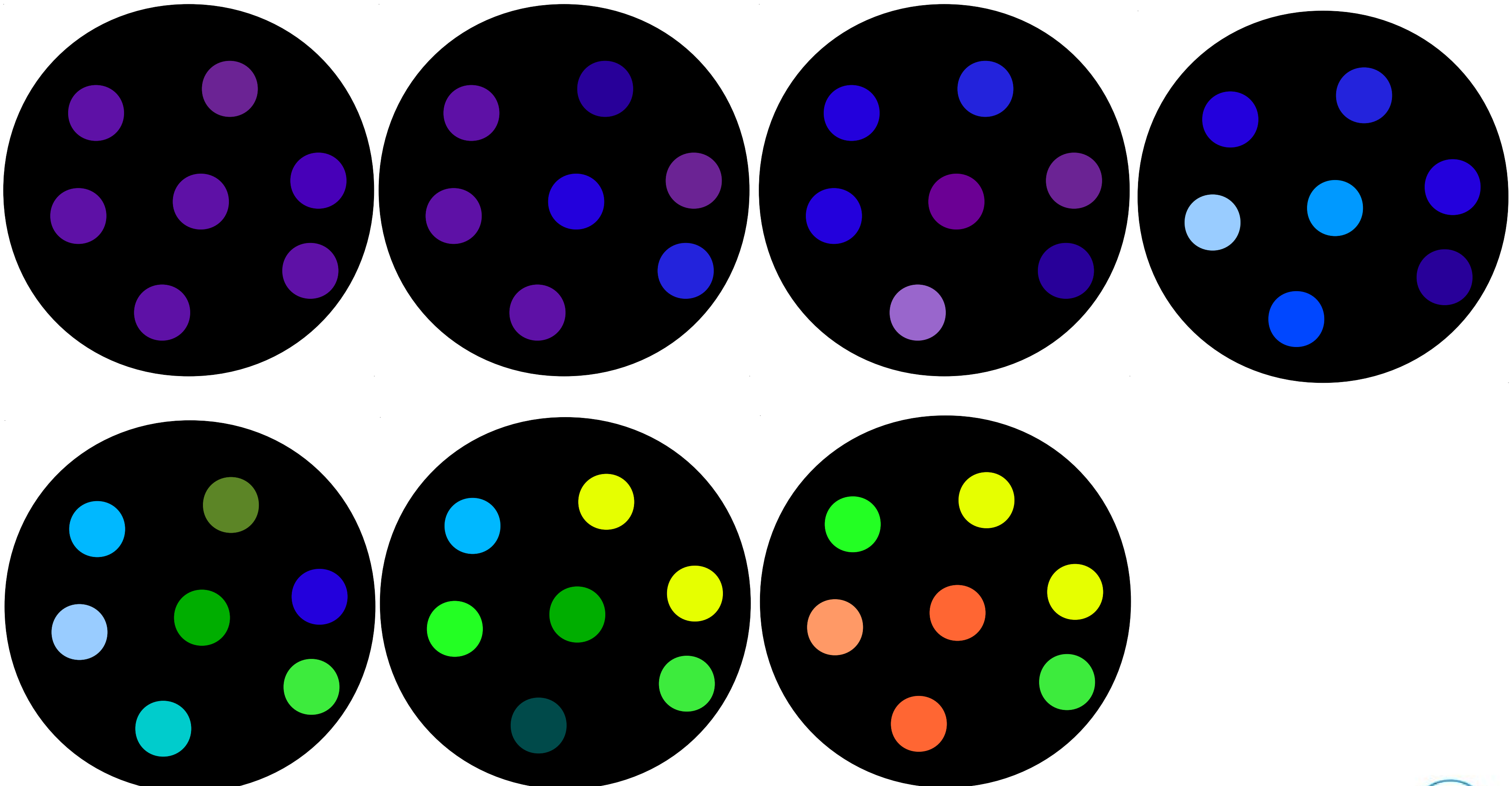
Evolution of solutions



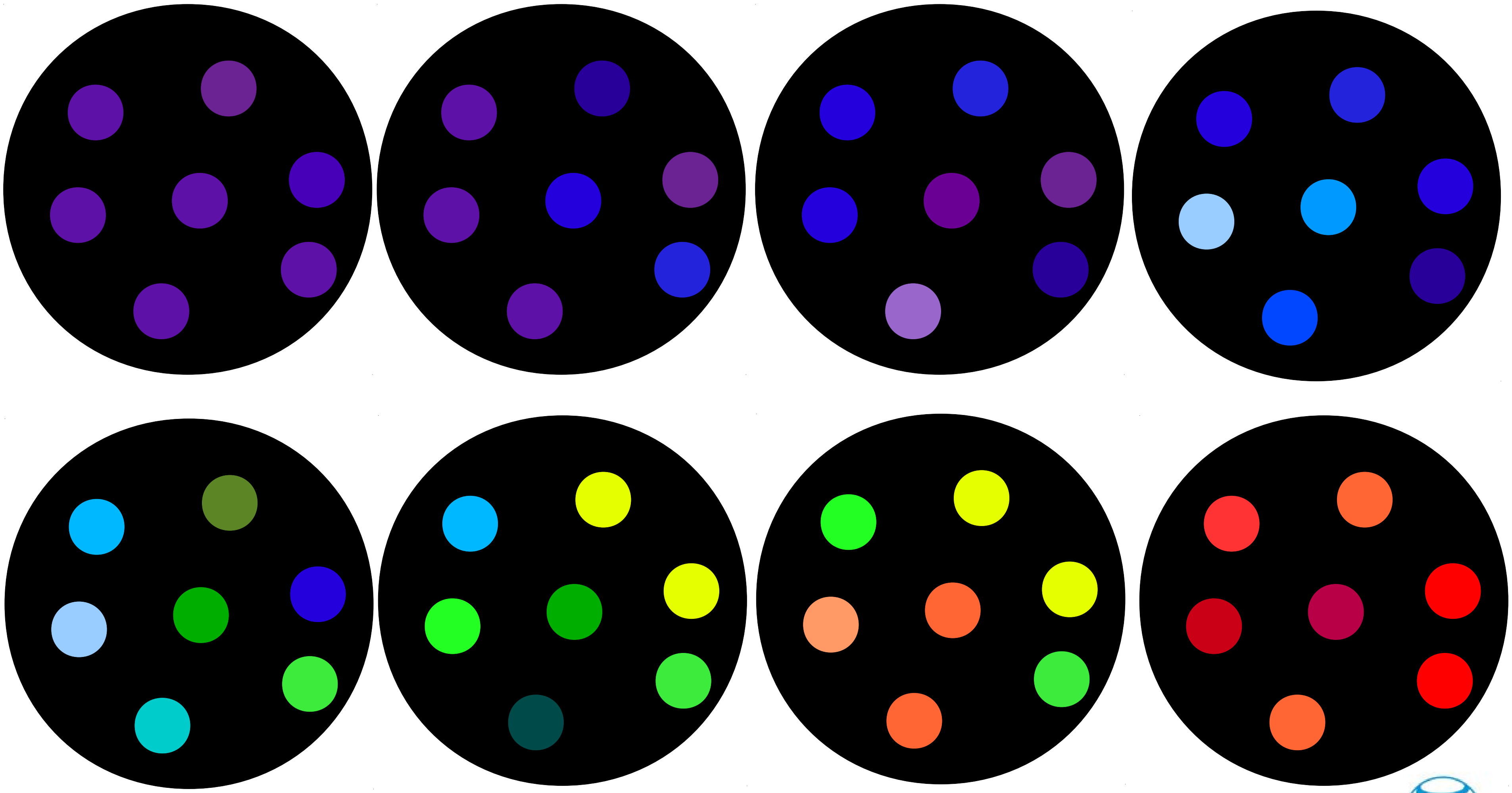
Evolution of solutions



Evolution of solutions



Evolution of solutions



Encoding solutions with random keys

Encoding with random keys

- A random key is a real random number in the continuous interval [0,1).

Encoding with random keys

- A random key is a real random number in the continuous interval [0,1).
- A vector X of random keys, or simply random keys, is an array of n random keys.

Encoding with random keys

- A random key is a real random number in the continuous interval [0,1).
- A vector X of random keys, or simply random keys, is an array of n random keys.
- Solutions of optimization problems can be encoded by random keys.

Encoding with random keys

- A random key is a real random number in the continuous interval [0,1).
- A vector X of random keys, or simply random keys, is an array of n random keys.
- Solutions of optimization problems can be encoded by random keys.
- A decoder is a deterministic algorithm that takes a vector of random keys as input and outputs a solution of the optimization problem.

Encoding with random keys: Sequencing

Encoding

[1, 2, 3, 4, 5]

X = [0.099, 0.216, 0.802, 0.368, 0.658]

Encoding with random keys: Sequencing

Encoding

[1, 2, 3, 4, 5]

X = [0.099, 0.216, 0.802, 0.368, 0.658]

Decode by sorting vector of random keys

[1, 2, 4, 5, 3]

X = [0.099, 0.216, 0.368, 0.658, 0.802]

Encoding with random keys: Sequencing

Therefore, the vector of random keys:

X = [0.099, 0.216, 0.802, 0.368, 0.658]

encodes the sequence: 1 – 2 – 4 – 5 – 3

Encoding with random keys: Subset selection (select 3 of 5 elements)

Encoding

[1, 2, 3, 4, 5]

X = [0.099, 0.216, 0.802, 0.368, 0.658]

Encoding with random keys: Subset selection (select 3 of 5 elements)

Encoding

[1, 2, 3, 4, 5]

X = [0.099, 0.216, 0.802, 0.368, 0.658]

Decode by sorting vector of random keys

[1, 2, 4, 5, 3]

X = [0.099, 0.216, 0.368, 0.658, 0.802]

Encoding with random keys: Subset selection (select 3 of 5 elements)

Therefore, the vector of random keys:

X = [0.099, 0.216, 0.802, 0.368, 0.658]

encodes the subset: {1, 2, 4 }

Encoding with random keys: Assigning integer weights ∈ [0,10] to a subset of 3 of 5 elements

Encoding

[	1,	2,	3,	4,	5 |	1,	2,	3,	4,	5]
X = [	0.099,	0.216,	0.802,	0.368,	0.658 |	0.4634,	0.5611,	0.2752,	0.4874,	0.0348]

Encoding with random keys: Assigning integer weights $\in [0,10]$ to a subset of 3 of 5 elements

Encoding

[1, 2, 3, 4, 5 | 1, 2, 3, 4, 5]

X = [0.099, 0.216, 0.802, 0.368, 0.658 | 0.4634, 0.5611, 0.2752, 0.4874, 0.0348]

Decode by sorting the first 5 keys and assign as the weight the value W_i = **floor** $[10 X_{5+i}] + 1$ to the 3 elements with smallest keys X_i, for $i = 1,\ldots,5$.

Encoding with random keys: Assigning integer weights $\in [0,10]$ to a subset of 3 of 5 elements

Therefore, the vector of random keys:

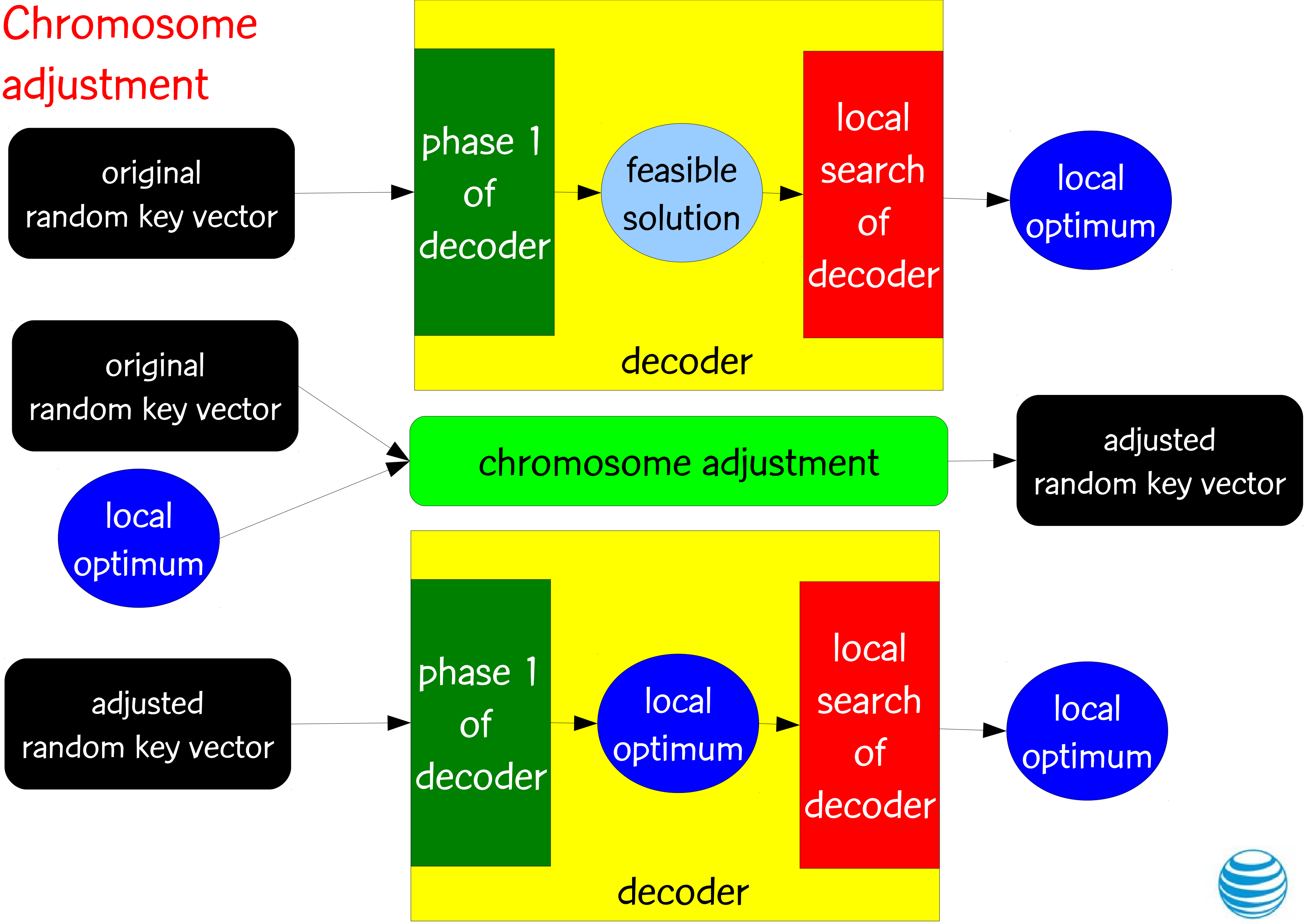
X = [0.099, 0.216, 0.802, 0.368, 0.658 | 0.4634, 0.5611, 0.2752, 0.4874, 0.0348]

encodes the weight vector W = (5,6,–,5,–)

Chromosome adjustment

Chromosome adjustment is useful in the case of complex decoders, e.g. those which have a local search module.

Chromosome adjustment

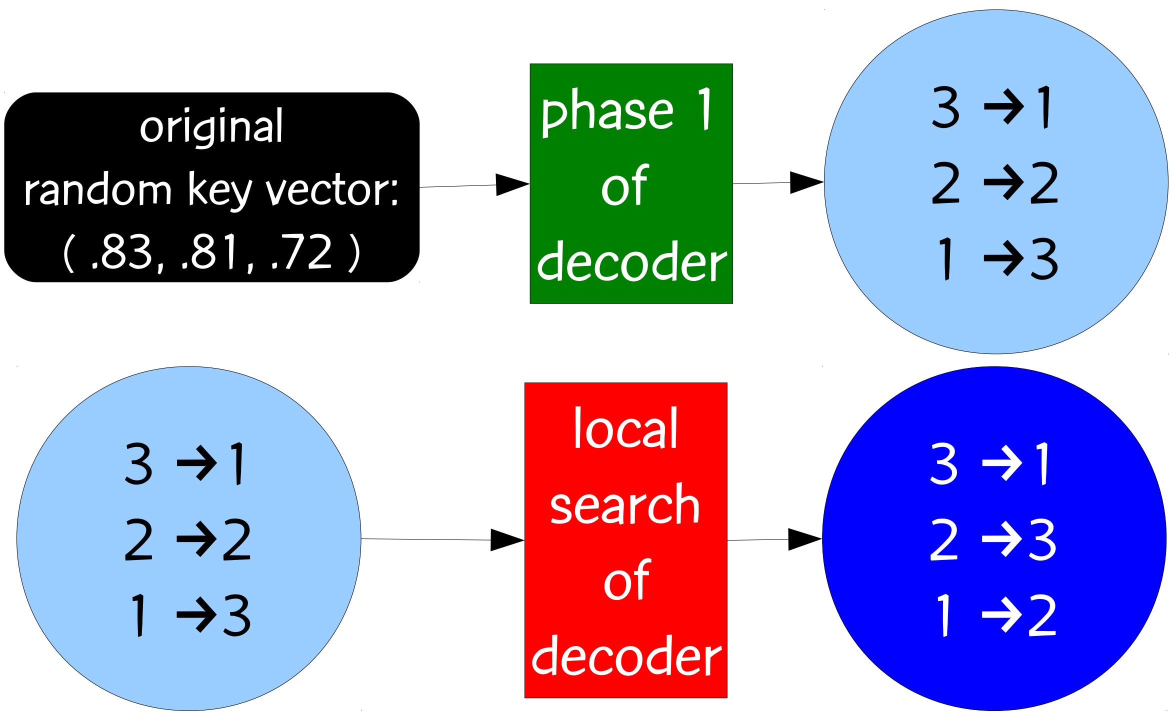


Quadratic assignment problem

Chromosome adjustment

Flow	1	2	3
1		20	30
2	20		40
3	30	40	

Dist	1	2	3
1		10	1
2	10		5
3	1	5	



cost = f(1,2) × d(3,2) +
f(1,3) × d(3,1) +
f(2,3) × d(2,1) =
100 + 30 + 400 = 530

local search swapped locations of facilities 1 and 2, resulting in cost = 200

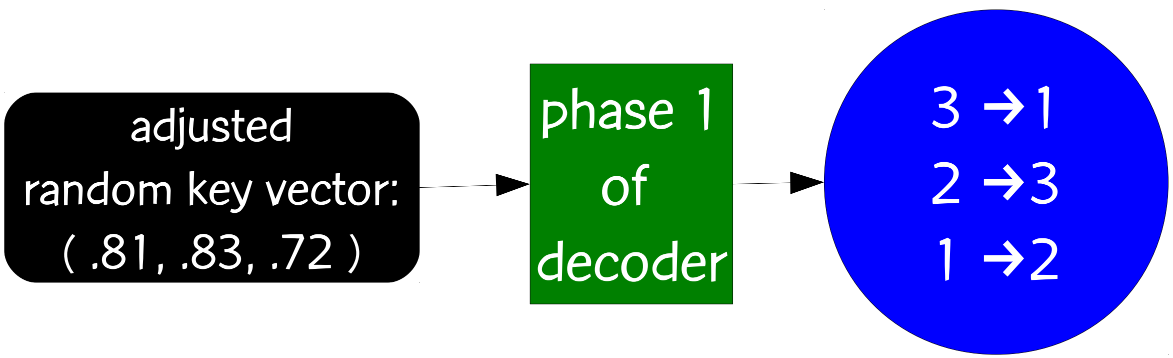
adjusted
random key vector:
(.81, .83, .72)

Quadratic assignment problem

Chromosome adjustment

Flow	1	2	3
1		20	30
2	20		40
3	30	40	

Dist	1	2	3
1		10	1
2	10		5
3	1	5	



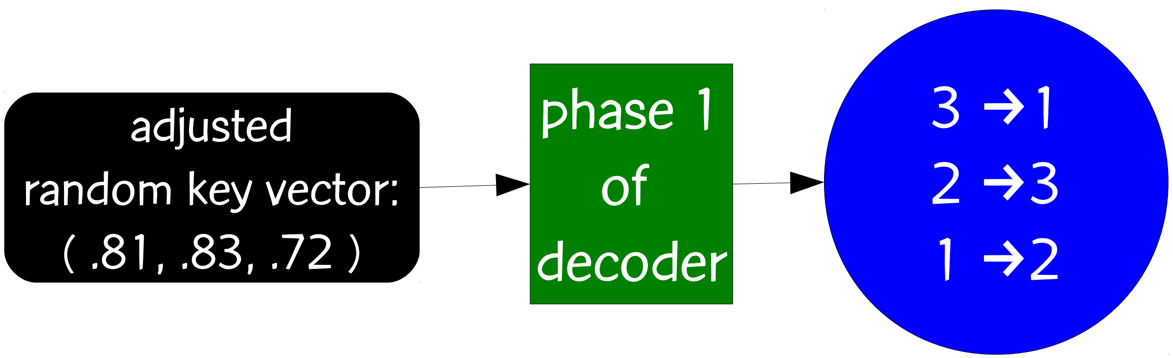
$$\text{cost} = f(1,2) \times d(2,3) + f(1,3) \times d(2,1) + f(2,3) \times d(3,1) = 100 + 60 + 40 = 200$$

Quadratic assignment problem

Chromosome adjustment

Flow	1	2	3
1		20	30
2	20		40
3	30	40	

Dist	1	2	3
1		10	1
2	10		5
3	1	5	



cost = f(1,2) × d(2,3) +
f(1,3) × d(2,1) +
f(2,3) × d(3,1) =
100 + 60 + 40 = 200

Not only is expensive local search avoided ...
Characteristics of local optimum are passed on to future generations .... They will be represented in the population by adjusted random key vector.

Genetic algorithms and 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 (key, or value of gene) 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 (key, or value of gene) 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 (key, or value of gene) 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 (key, or value of gene) 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 (key, or value of gene) 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 (key, or value of gene) 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 (key, or value of gene) 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

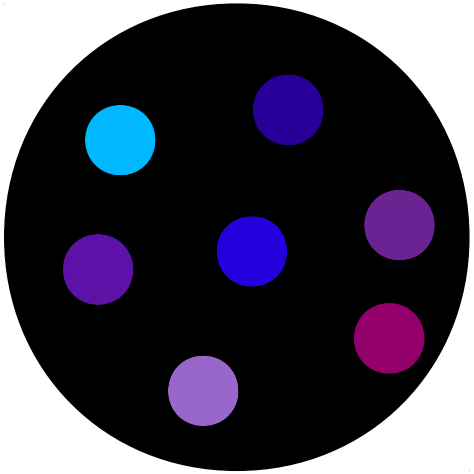
- Mating is done using parametrized uniform crossover (Spears & DeJong , 1990)
- For each gene, flip a biased coin to choose which parent passes the allele (key, or value of gene) to the child.

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

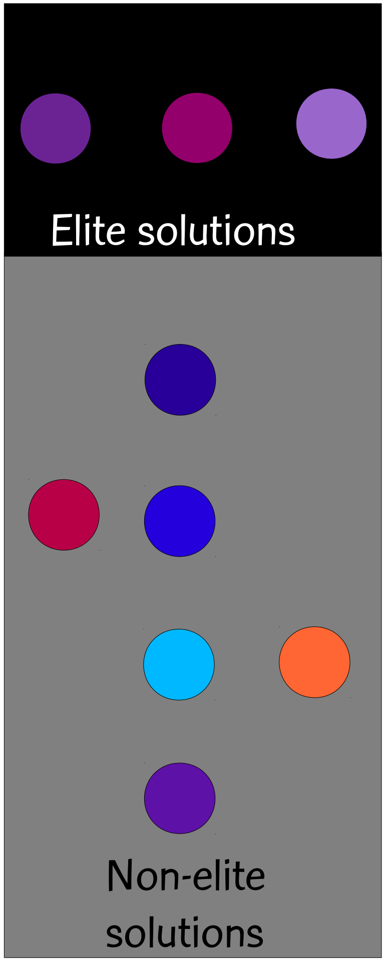
Initial population is made up of P random-key vectors, each with N keys, each having a value generated uniformly at random in the interval [0,1).



GAs and random keys

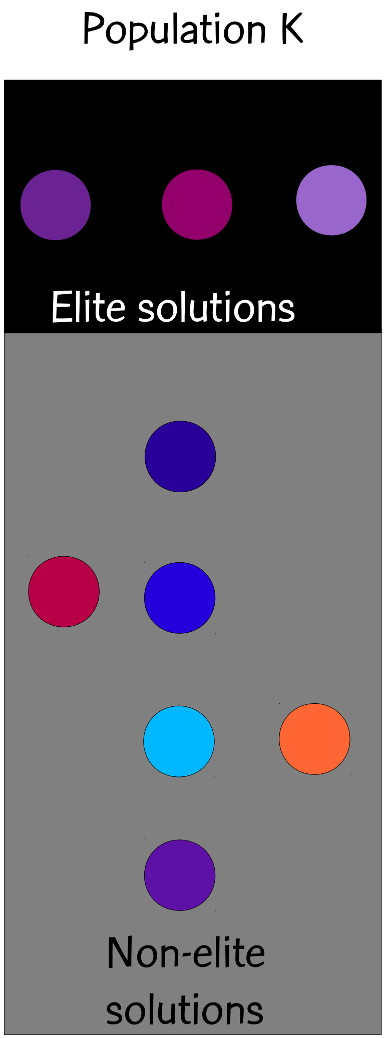
At the K-th generation, compute the cost of each solution ...

Population K



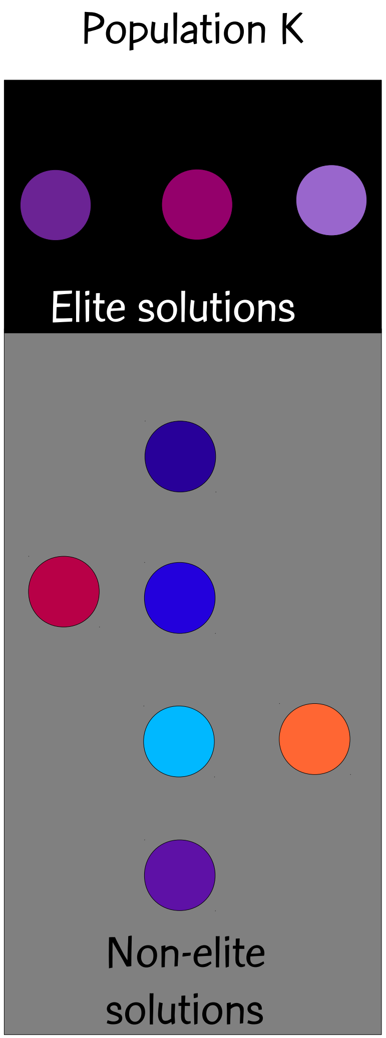
GAs and random keys

At the K-th generation, compute the cost of each solution and partition the solutions into two sets:



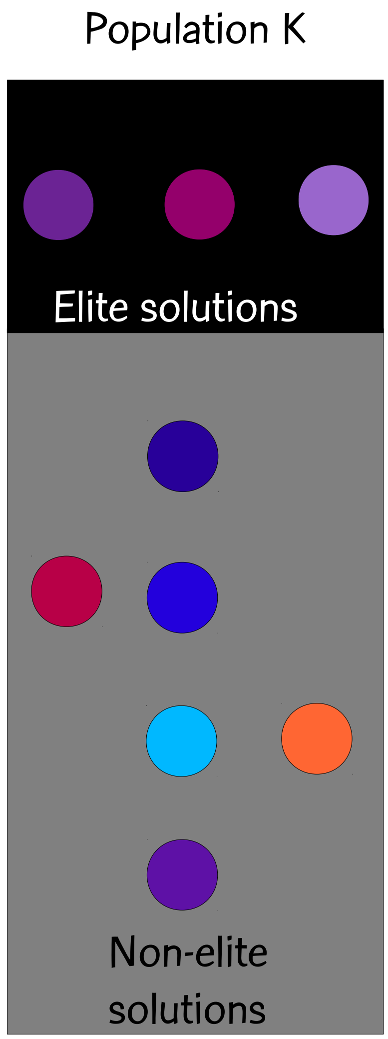
GAs and random keys

At the K-th generation, compute the cost of each solution and partition the solutions into two sets: elite solutions and non-elite solutions.



GAs and random keys

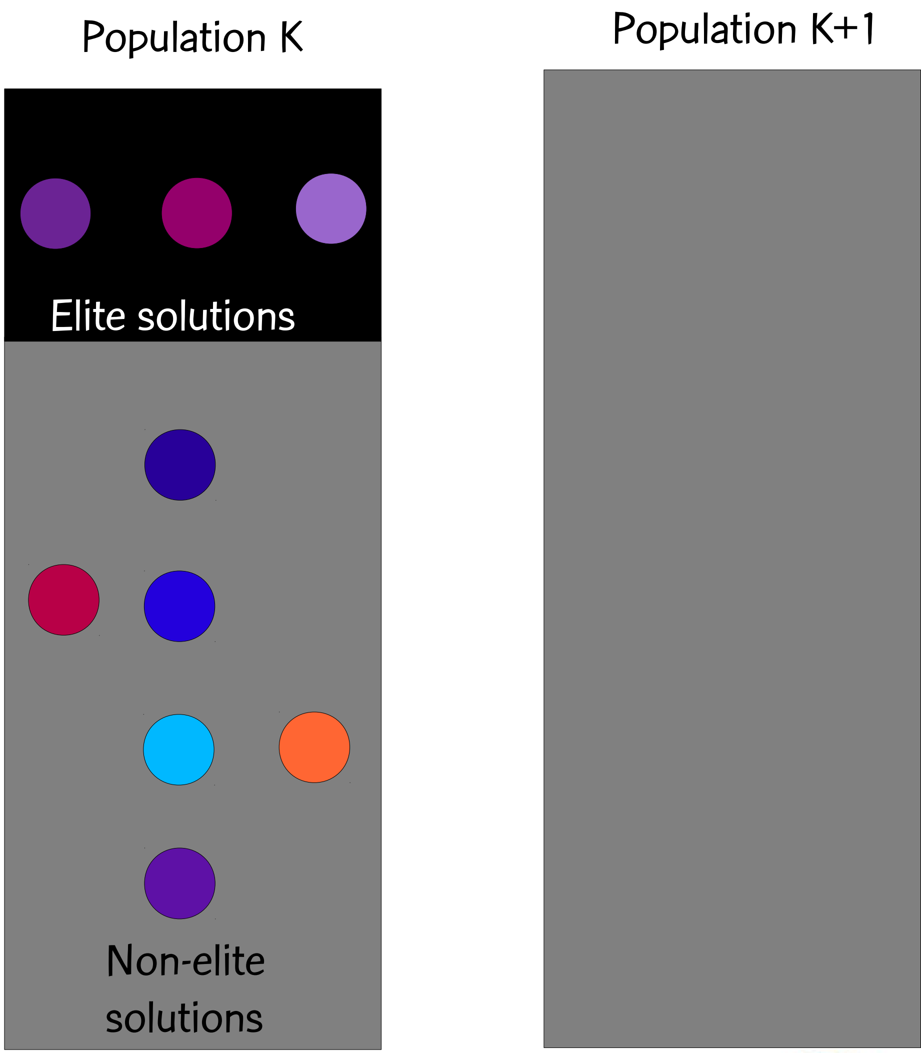
At the K-th generation, compute the cost of each solution and partition the solutions into two sets: elite solutions and non-elite solutions. Elite set should be smaller of the two sets and contain best solutions.



at&t Your world. Delivered.

GAs and random keys

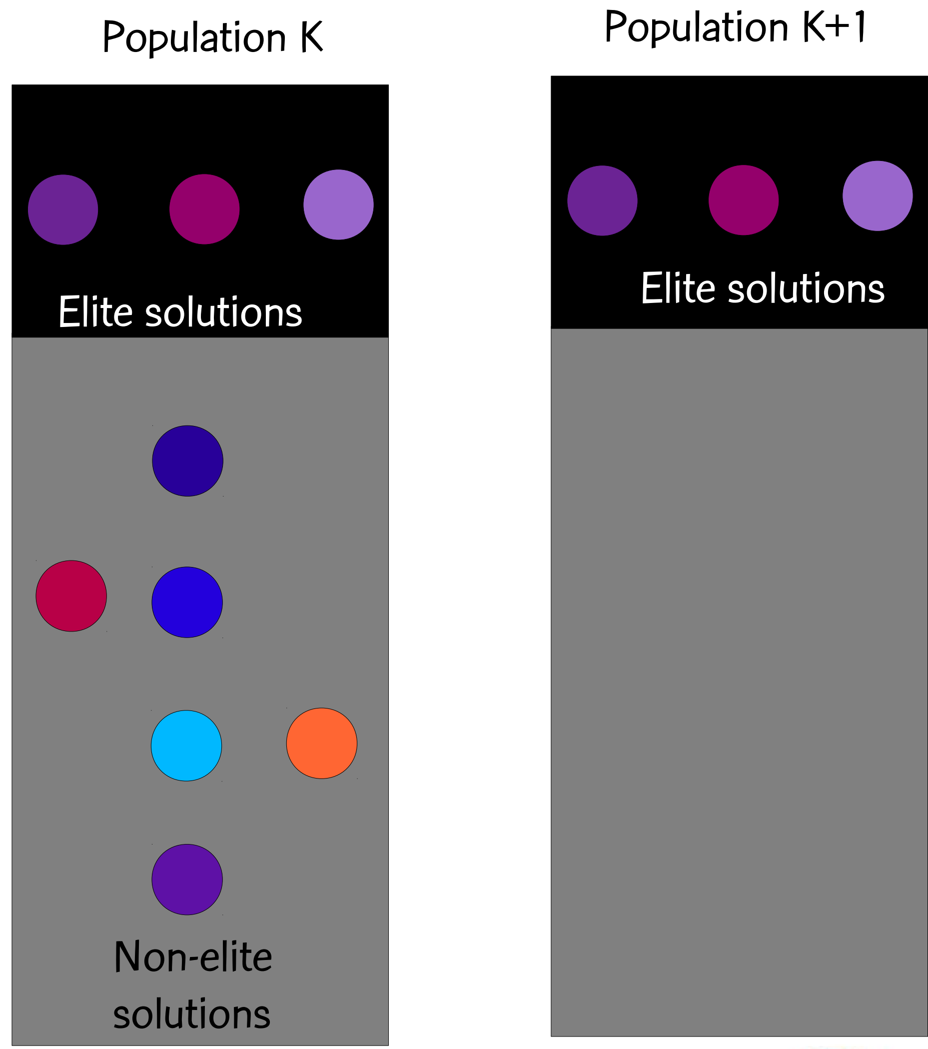
Evolutionary dynamics



GAs and random keys

Evolutionary dynamics

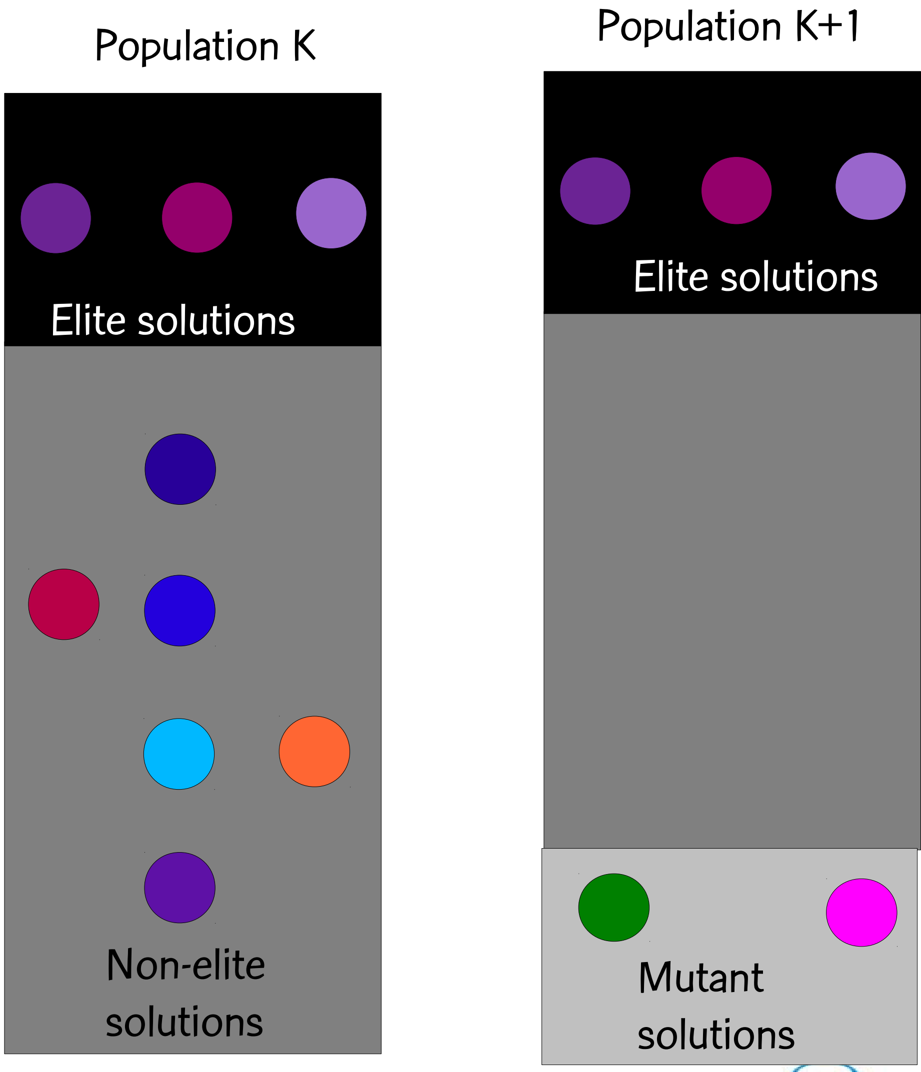
– Copy elite solutions from population K to population K+1



GAs and random keys

Evolutionary dynamics

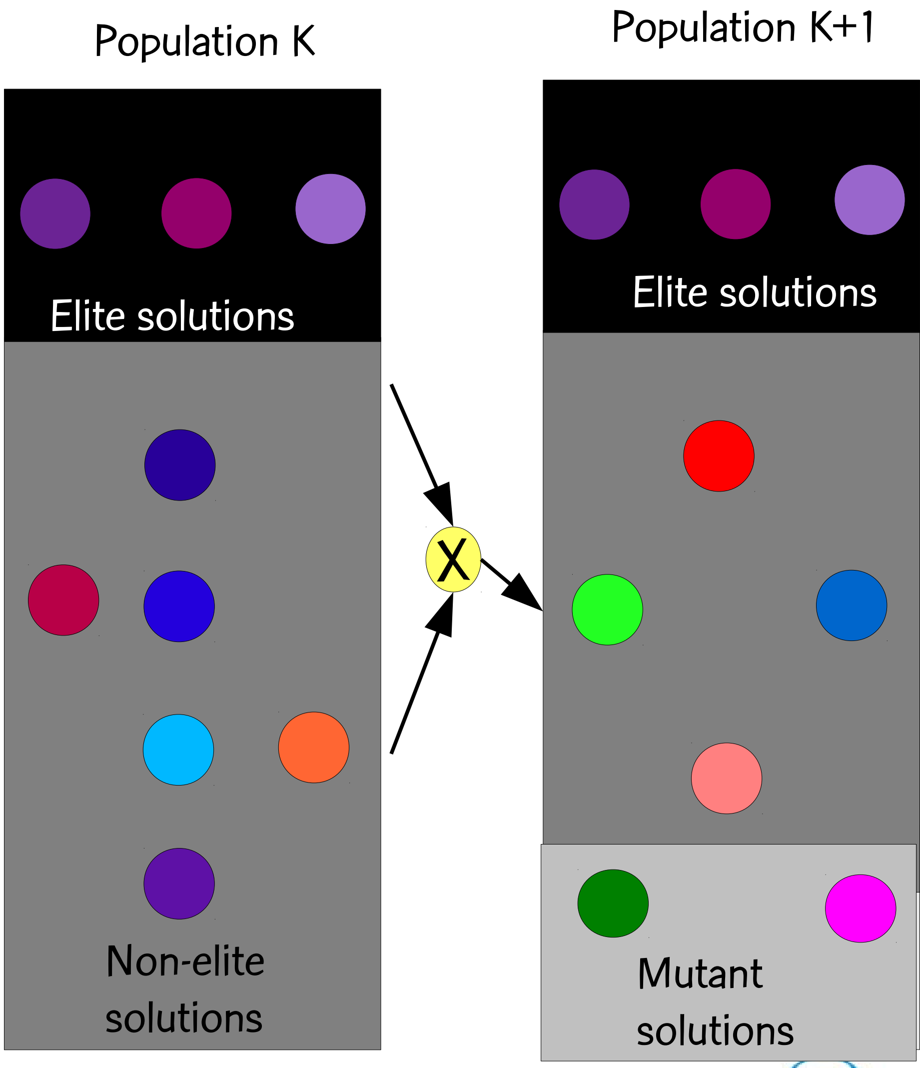
- Copy elite solutions from population K to population K+1
- Add R random solutions (mutants) to population K+1



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
- While K+1-th population < P
  - RANDOM-KEY GA: Use any two solutions in population K to produce child in population K+1. Mates are chosen at random.

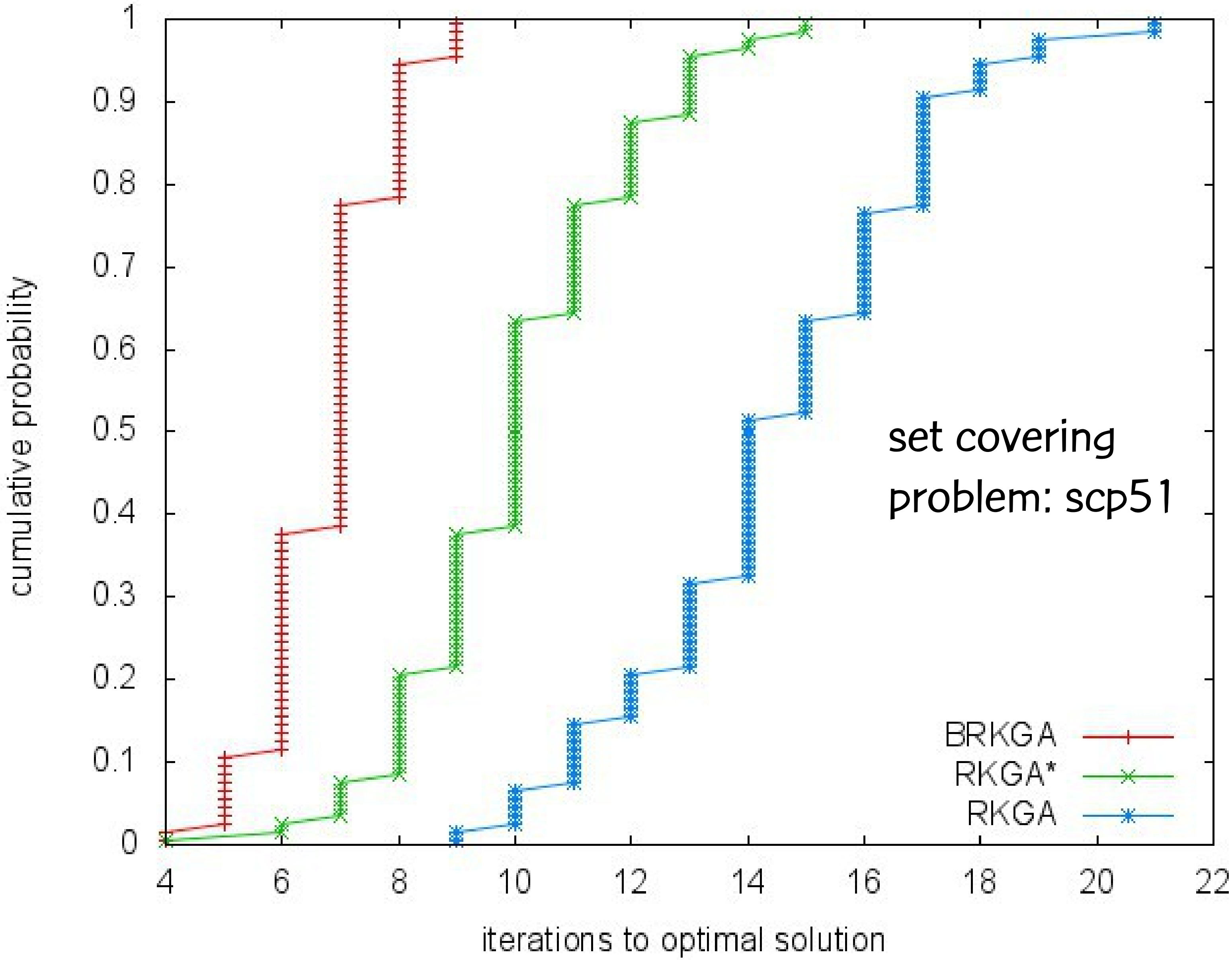


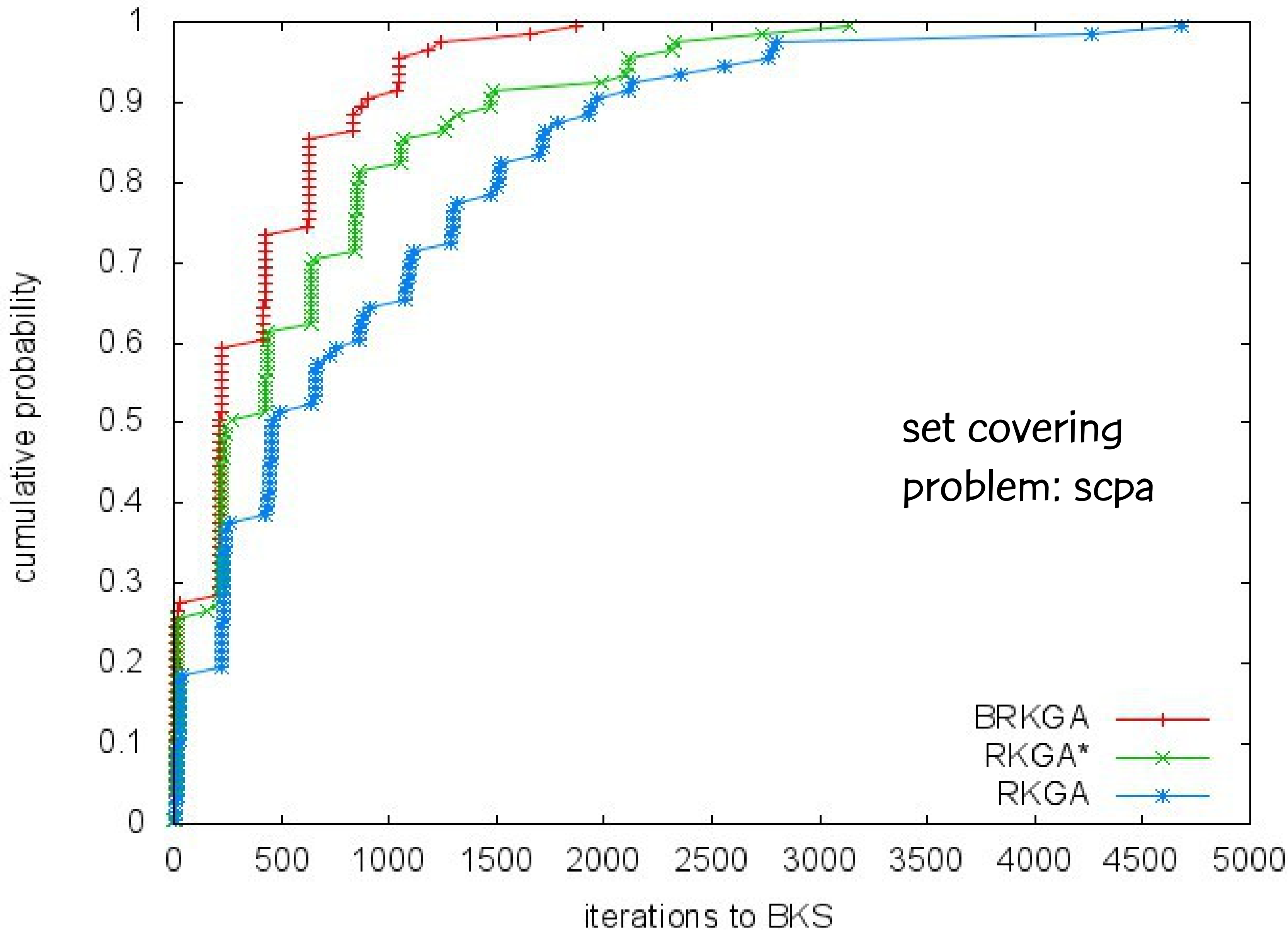
Biased random key genetic algorithm

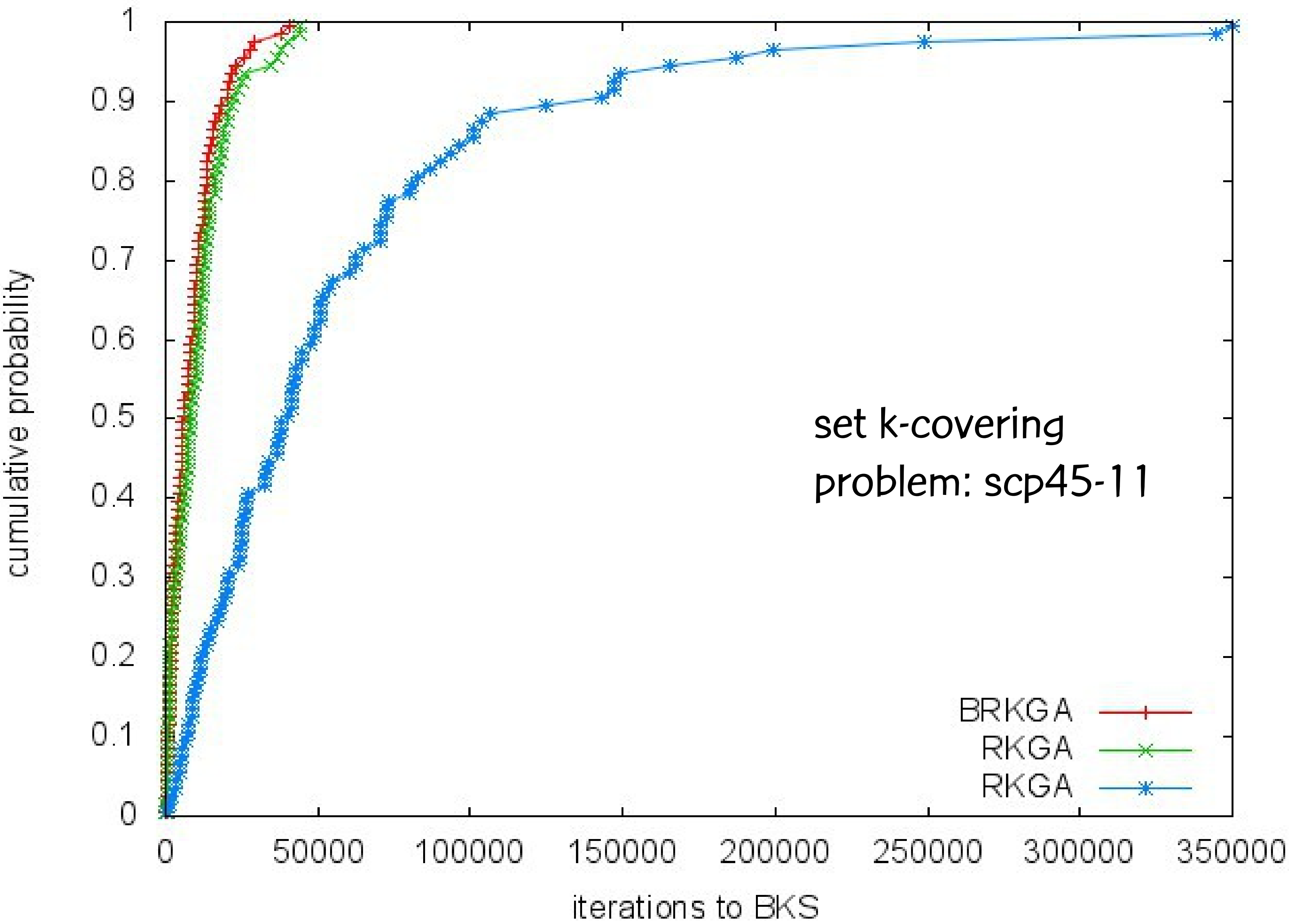
- A biased random key genetic algorithm (BRKGA) is a random key genetic algorithm (RKGA).
- BRKGA and RKGA differ in how mates are chosen for crossover and how parametrized uniform crossover is applied.

How RKGA & BRKGA differ

RKGA	**BRKGA**
both parents chosen at random from entire population	both parents chosen at random but one parent chosen from population of elite solutions
either parent can be parent A in parametrized uniform crossover	best fit parent is parent A in parametrized uniform crossover


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
cumulative probability
4
6
8
10
12
14
16
18
20
22
iterations to optimal solution
BRKGA
RKGA*
RKGA
set covering
problem: scp51


cumulative probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
iterations to BKS
set covering problem: scpa
BRKGA
RKGA*
RKGA


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
cumulative probability
0
50000
100000
150000
200000
250000
300000
350000
iterations to BKS
set k-covering
problem: scp45-11
BRKGA
RKGA
RKGA

Two types of parent selection in BRKGA

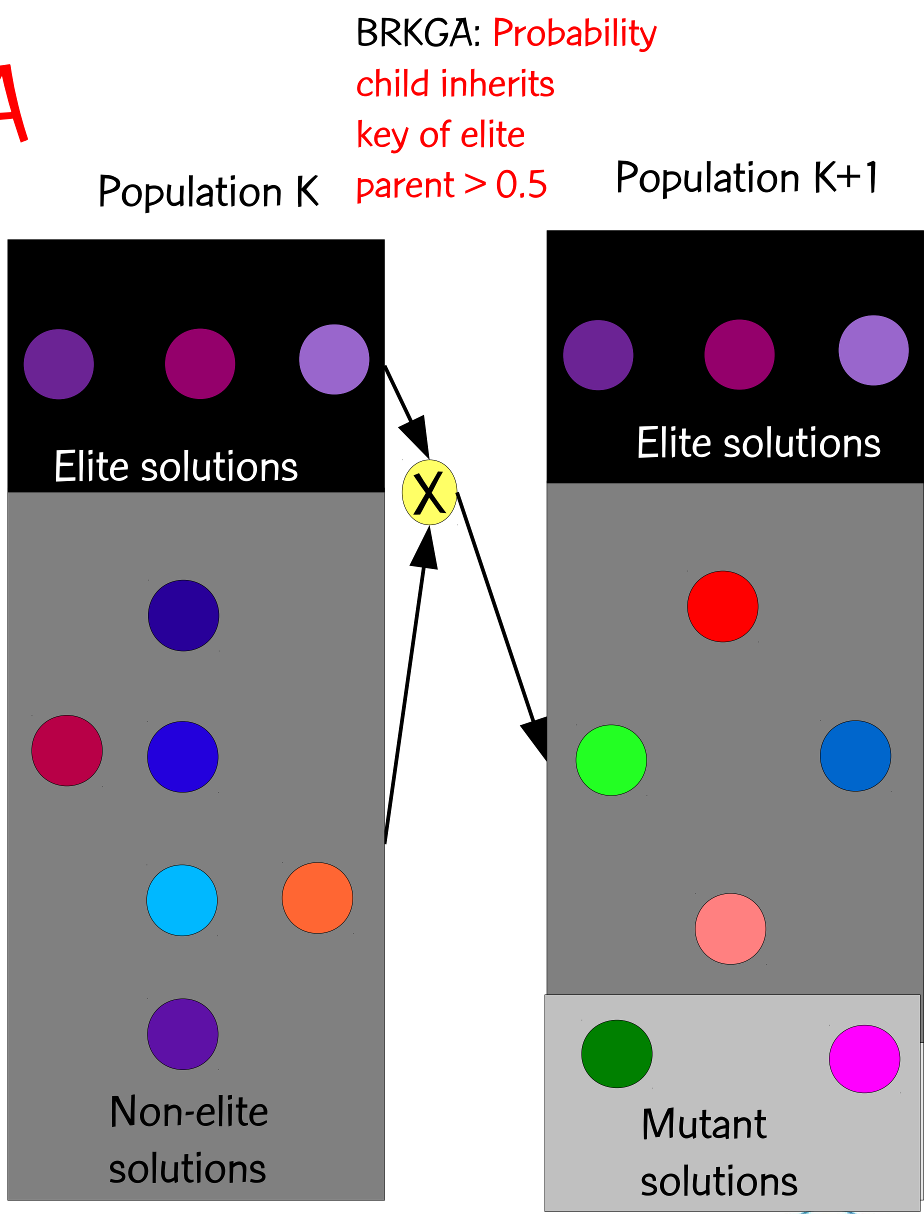
1) select second parent from population of non-elite solutions

2) select second parent from entire population, excluding the selected first parent

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
  - RANDOM-KEY GA: Use any two solutions in population K to produce child in population K+1. Mates are chosen at random.
  - BIASED RANDOM-KEY GA: Mate elite solution with other solution of population K to produce child in population K+1. Mates are chosen at random.



Observations

- Random method: keys are randomly generated so solutions are always vectors of random keys

Observations

- Random method: keys are randomly generated so solutions are always vectors of random keys
- Elitist strategy: best solutions are passed without change from one generation to the next (incumbent is kept)

Observations

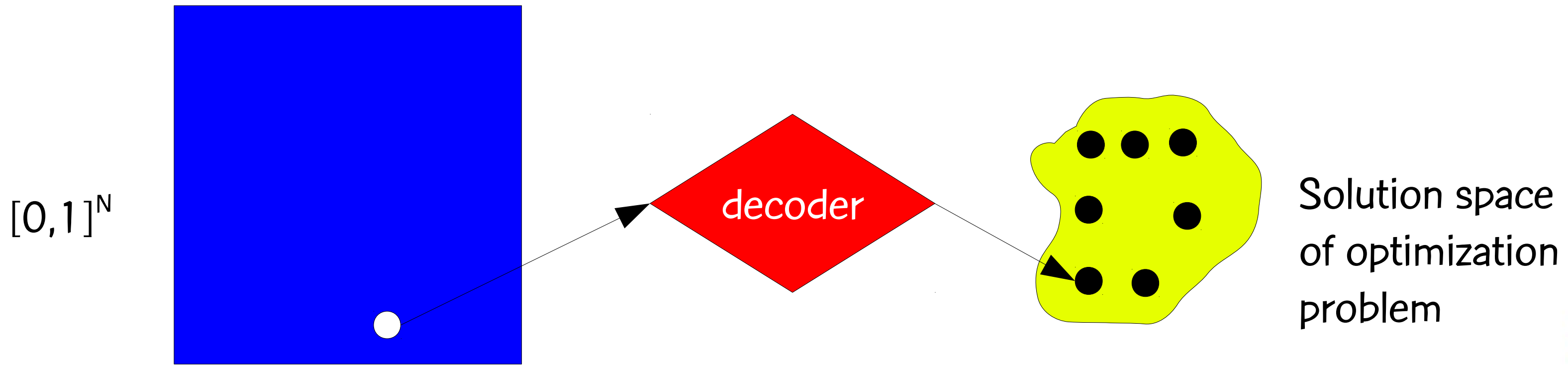
- Random method: keys are randomly generated so solutions are always vectors of random keys
- Elitist strategy: best solutions are passed without change from one generation to the next (incumbent is kept)
- Child inherits more characteristics of elite parent: one parent is always selected (with replacement) from the small elite set and probability that child inherits key of elite parent > 0.5 Not so in the RKGA of Bean.

Observations

- Random method: keys are randomly generated so solutions are always vectors of random keys
- Elitist strategy: best solutions are passed without change from one generation to the next (incumbent is kept)
- 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 Not so in the RKGA of Bean.
- No mutation in crossover: mutants are used instead (they play same role as mutation in GAs … help escape local optima)

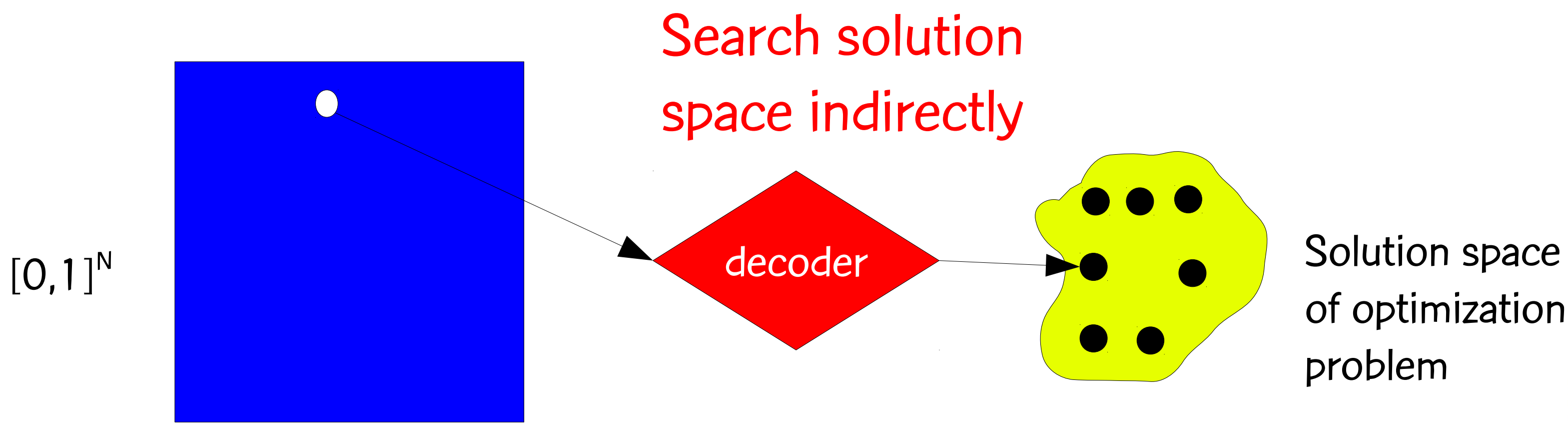
Decoders

- 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.



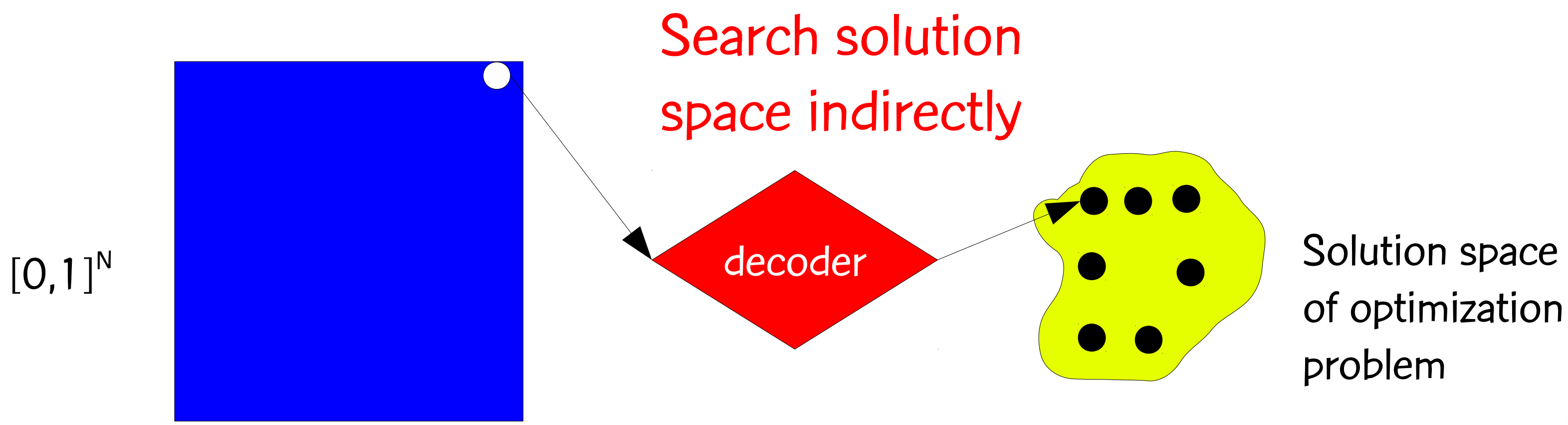
Decoders

- 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.



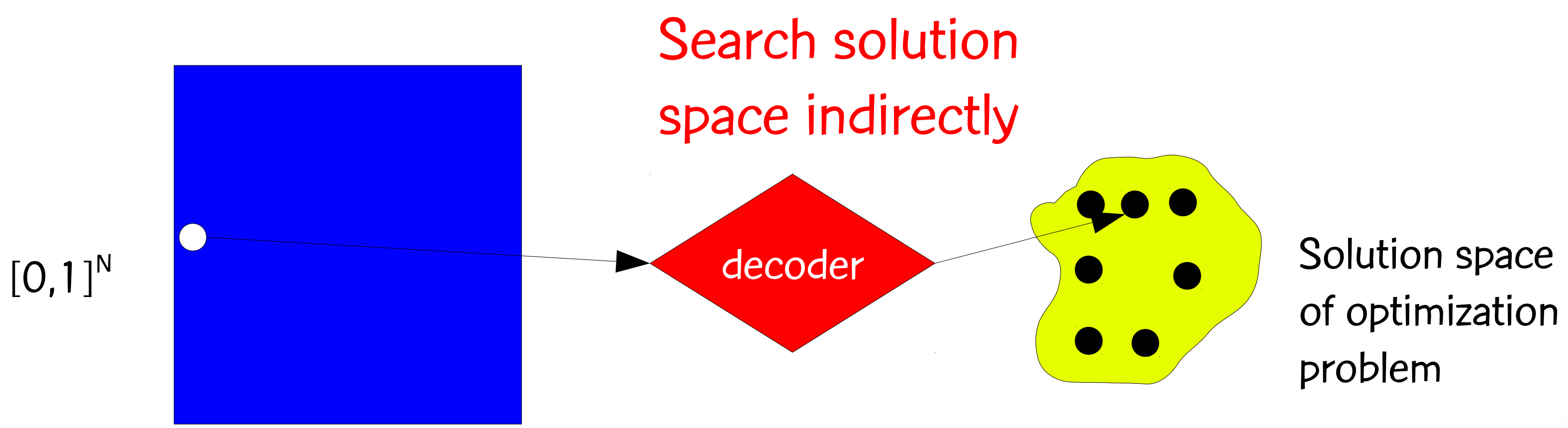
Decoders

- 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.

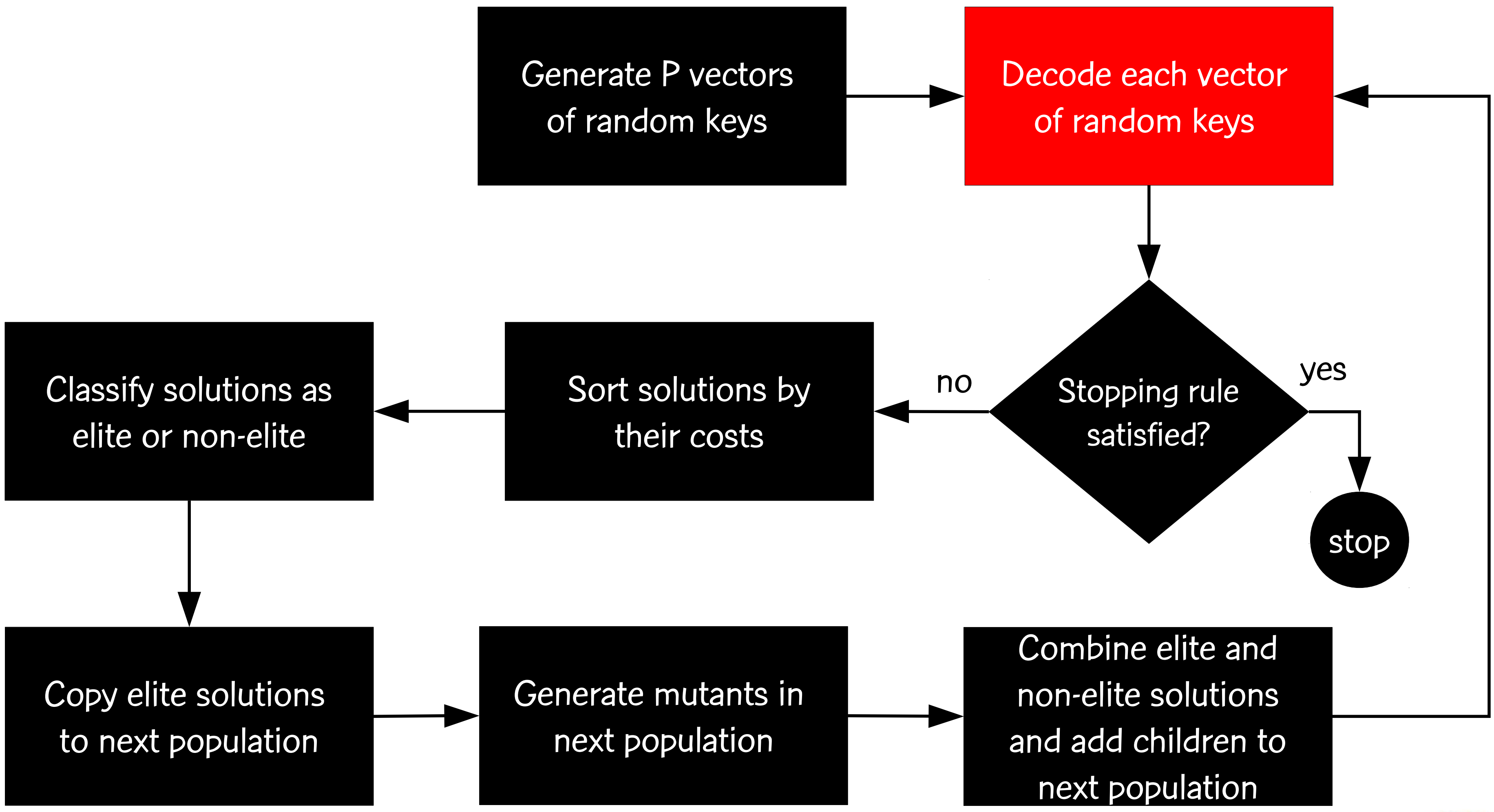


Decoders

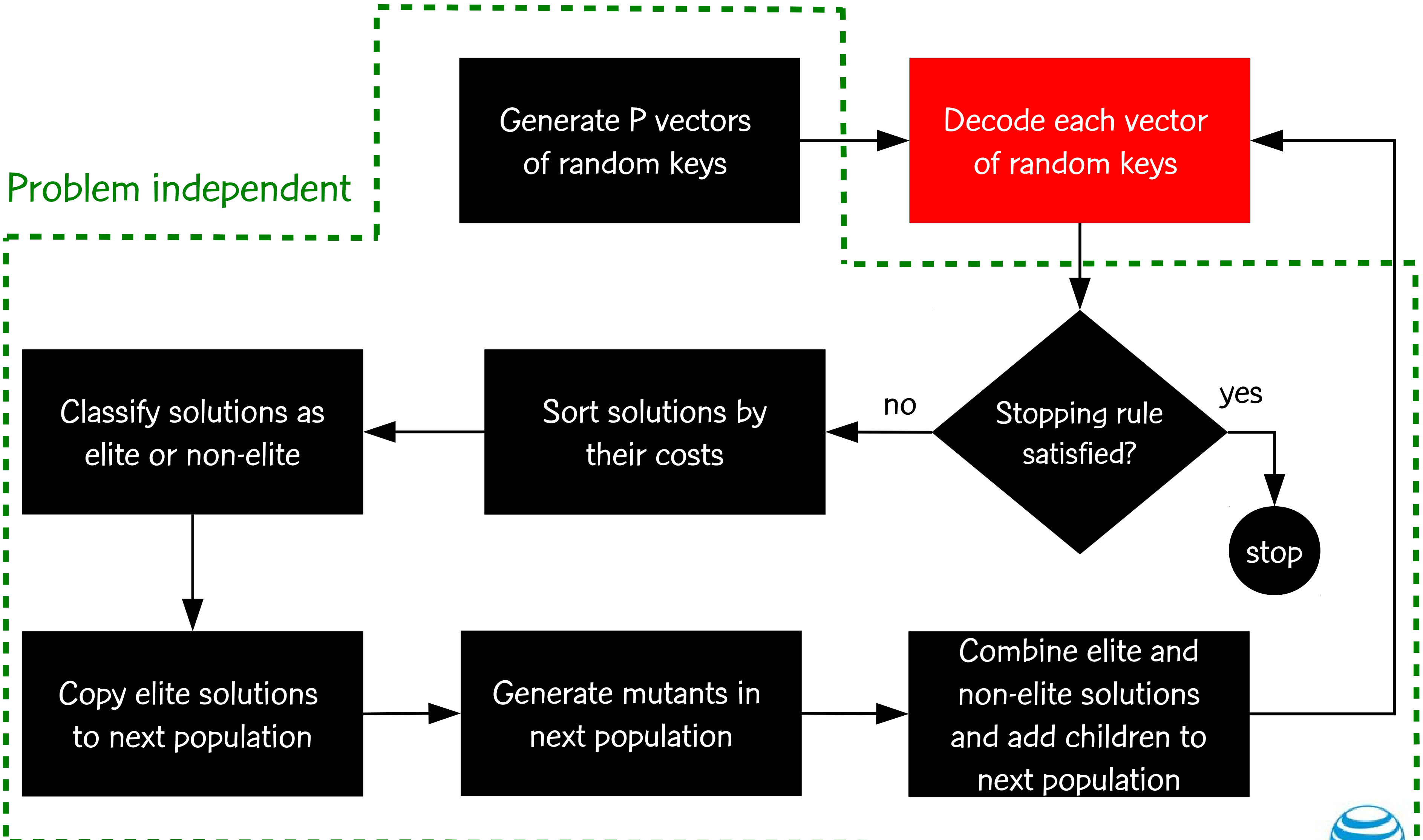
- A decoder is a deterministic algorithm that takes as input a random-key vector and returns a feasible solution of the optimization problem and its cost.
- Bean (1994) proposed decoders based on sorting the random-key vector to produce a sequence.
- A random-key GA searches the solution space indirectly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



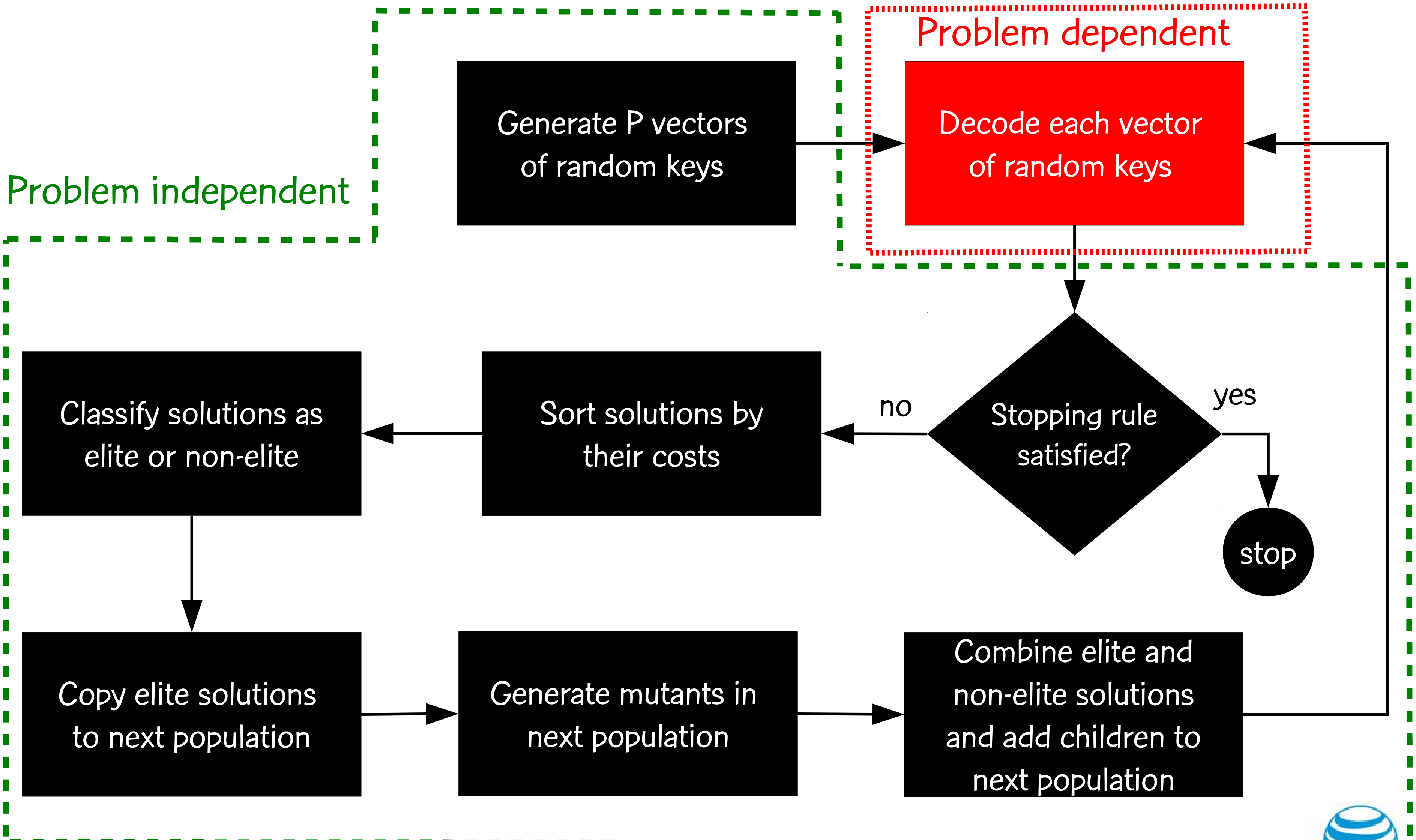
Framework for biased random-key genetic algorithms



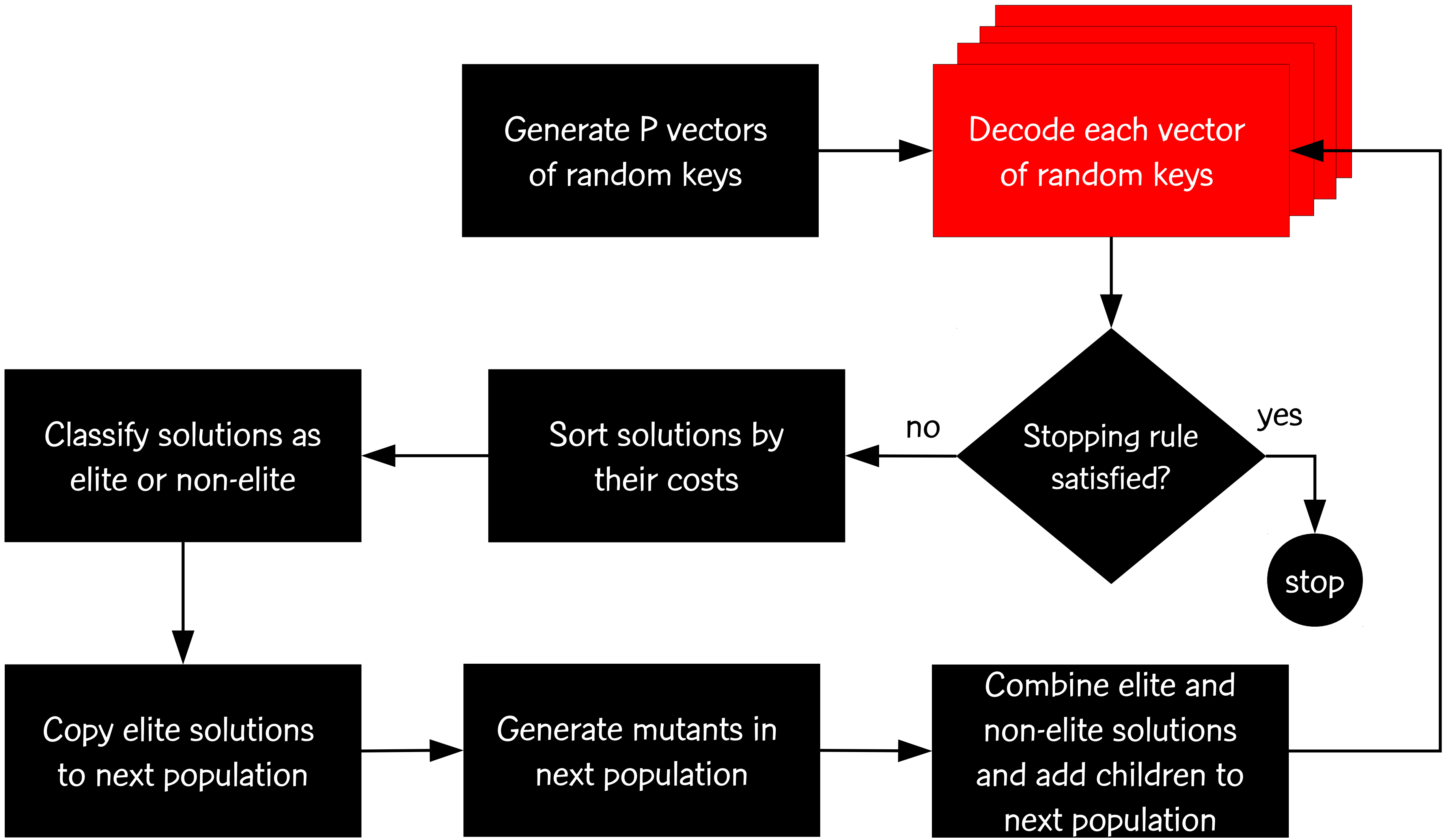
Framework for biased random-key genetic algorithms



Framework for biased random-key genetic algorithms

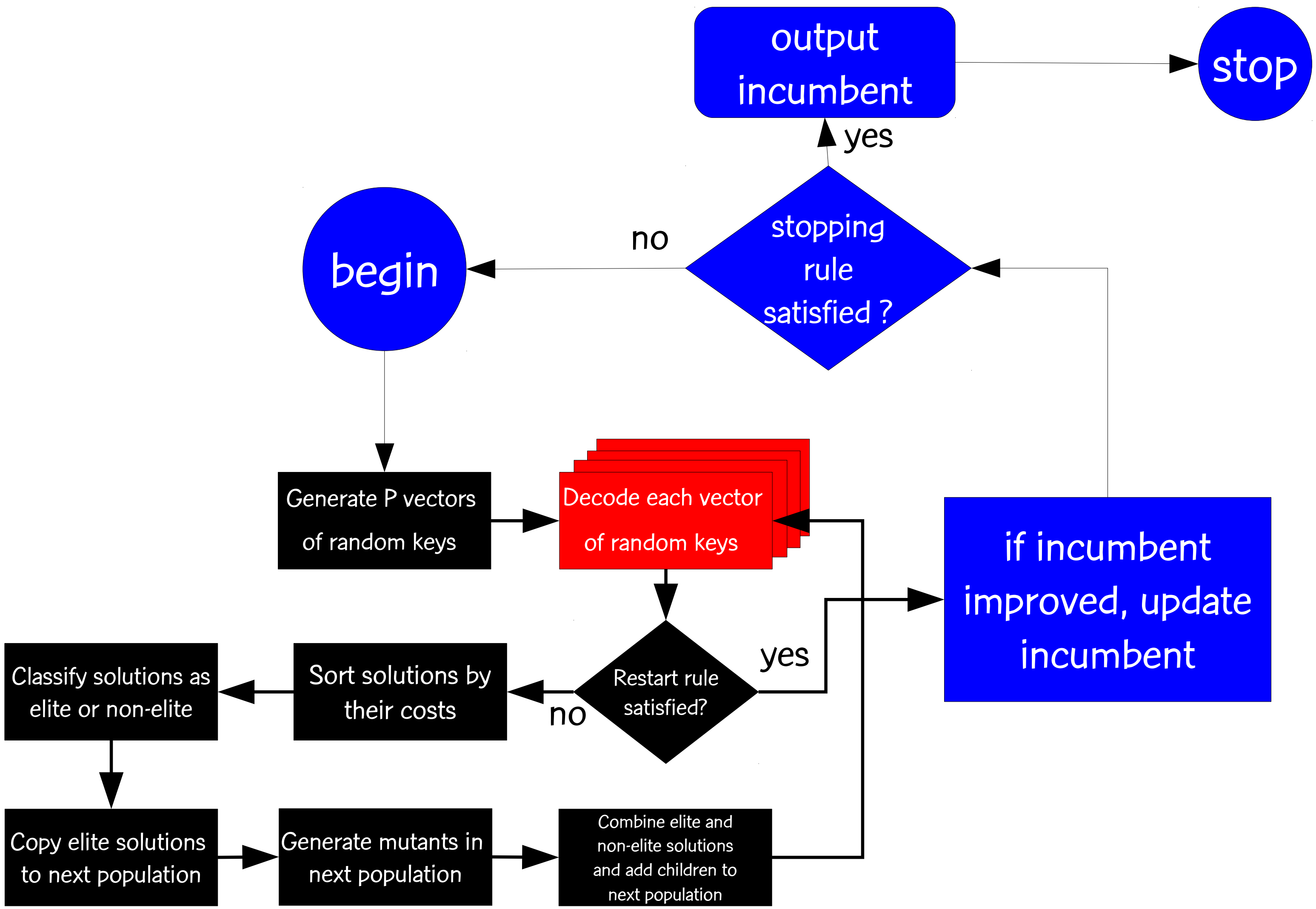


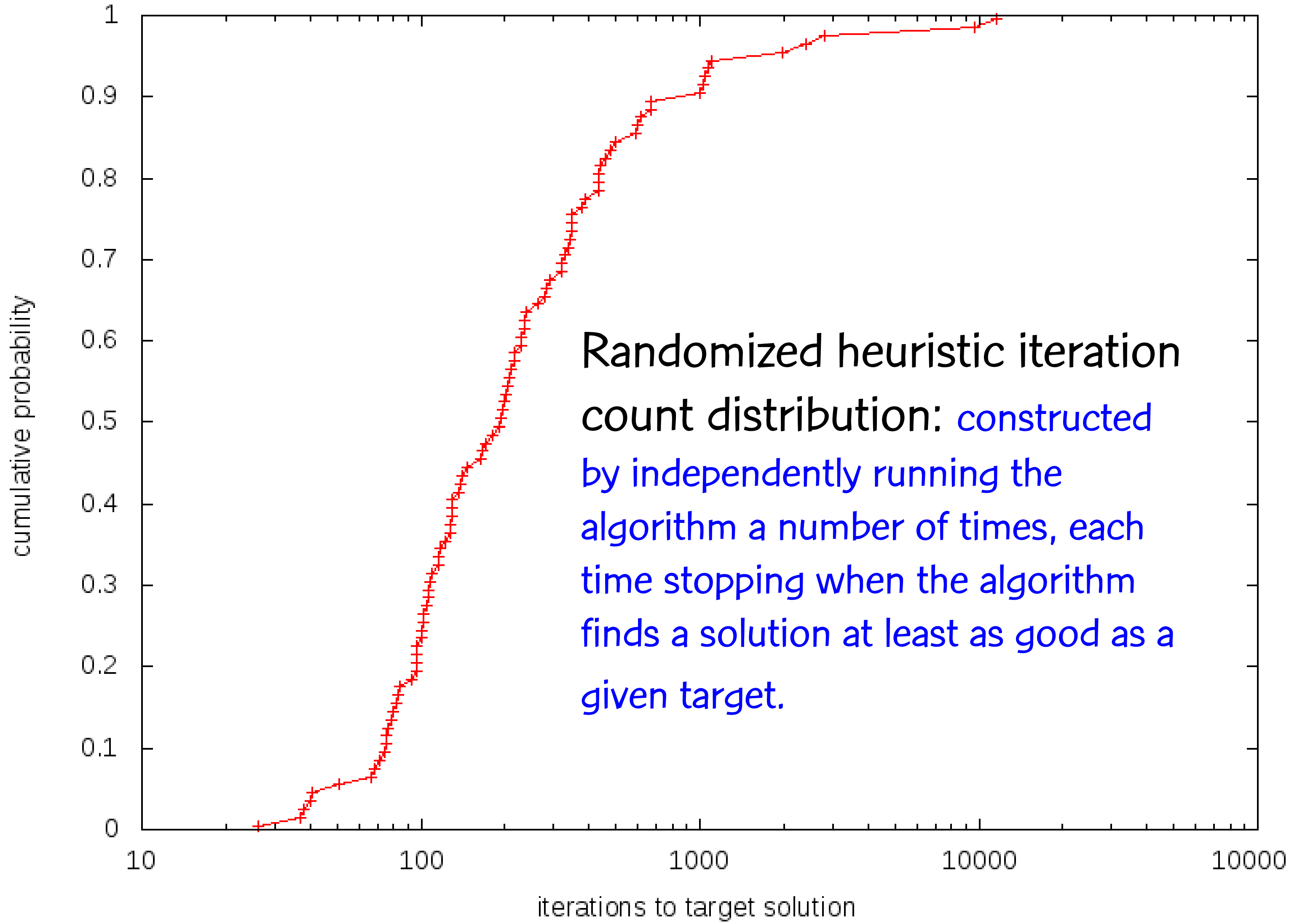
Decoding of random key vectors can be done in parallel



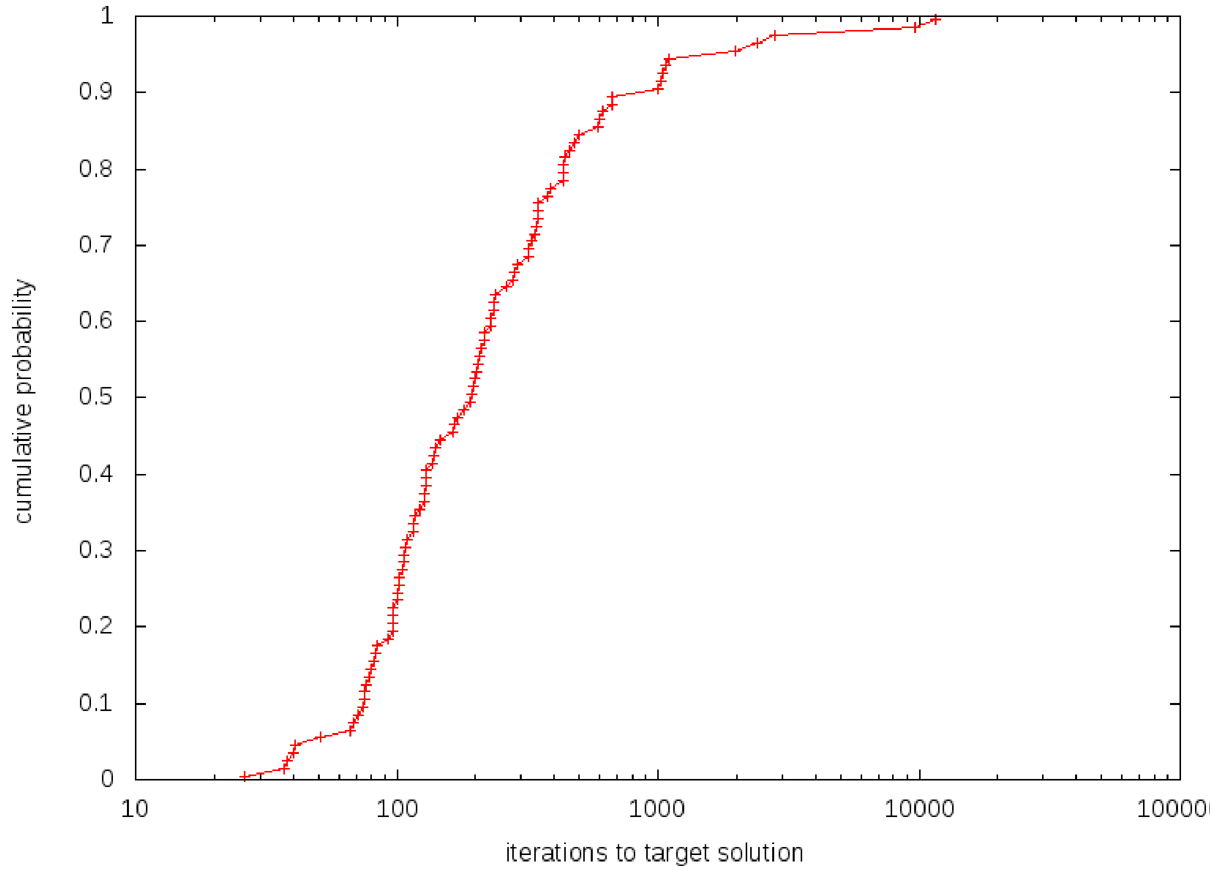
at&t
Your world. Delivered.

BRKGA in multi-start strategy

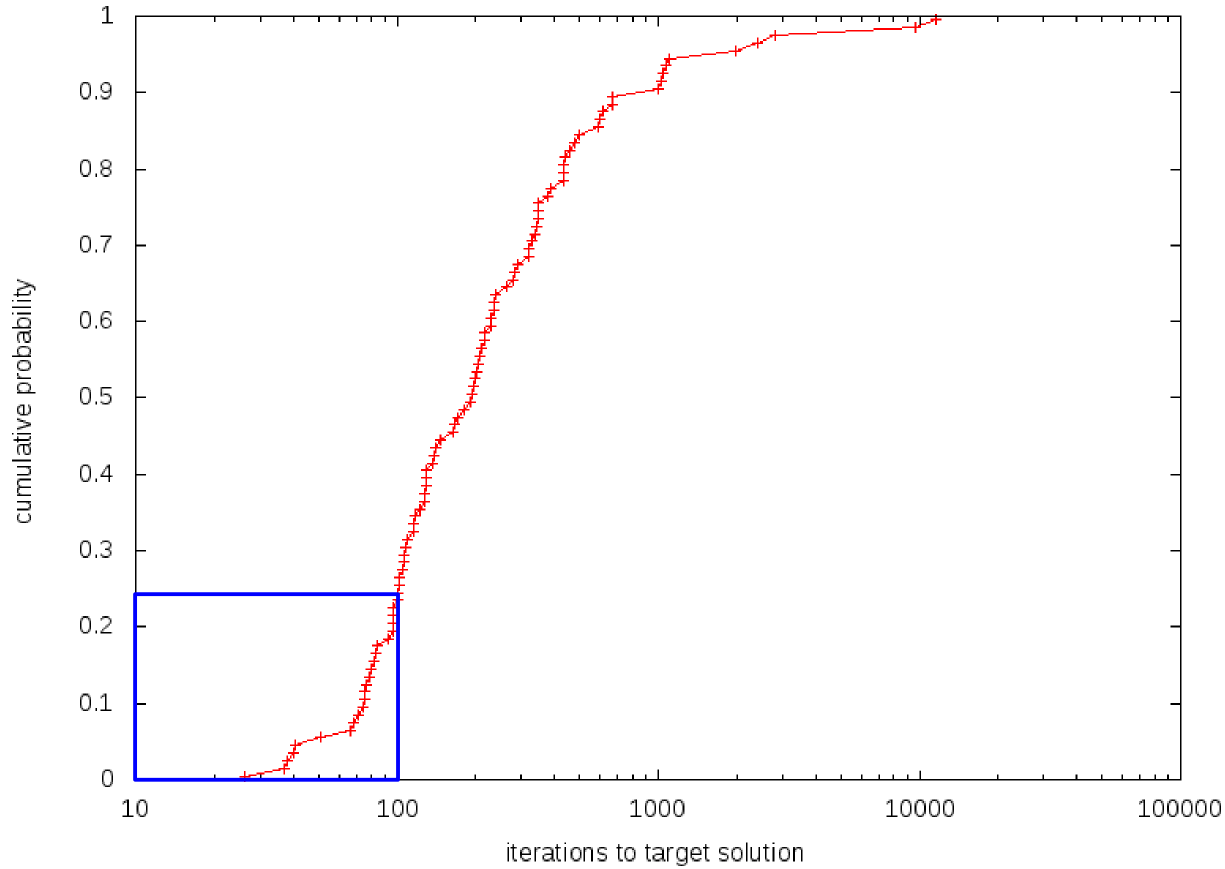




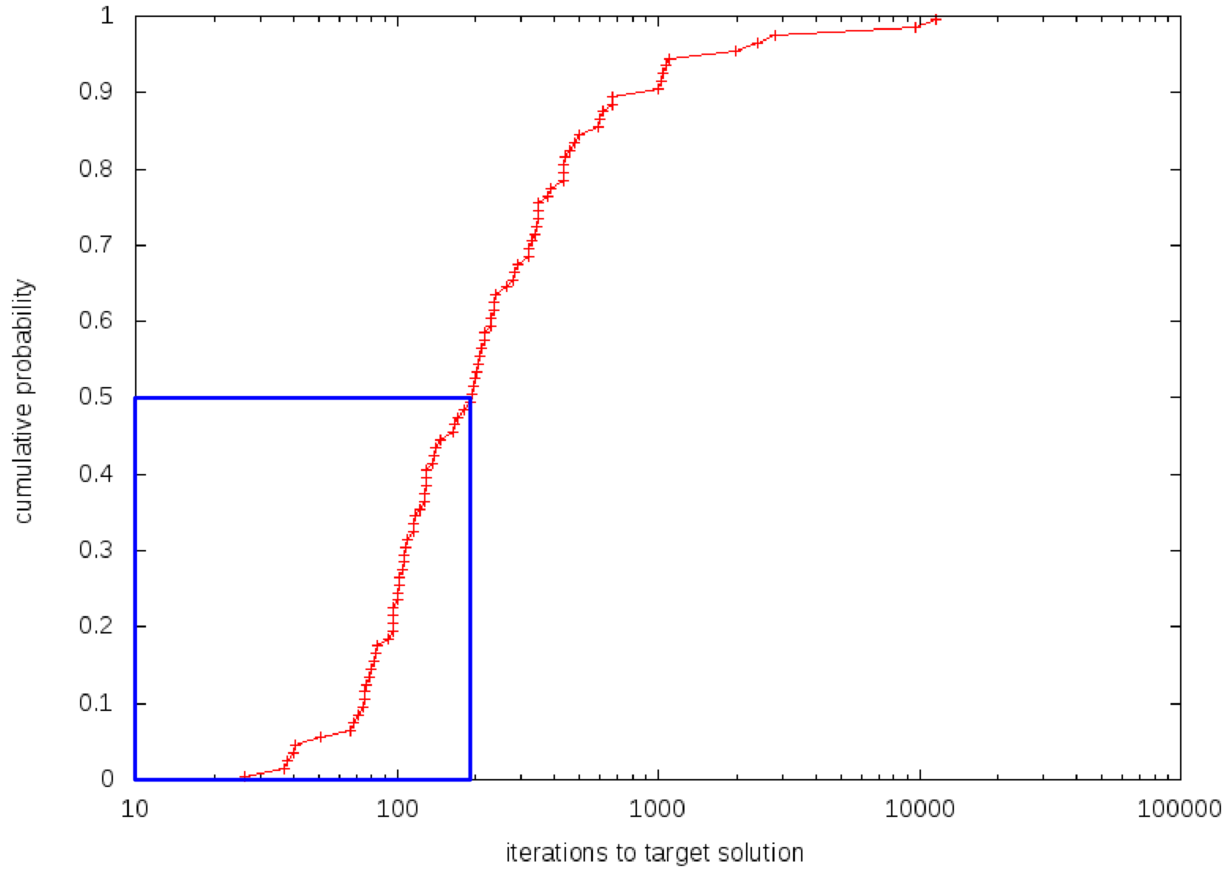
Randomized heuristic iteration count distribution: constructed by independently running the algorithm a number of times, each time stopping when the algorithm finds a solution at least as good as a given target.



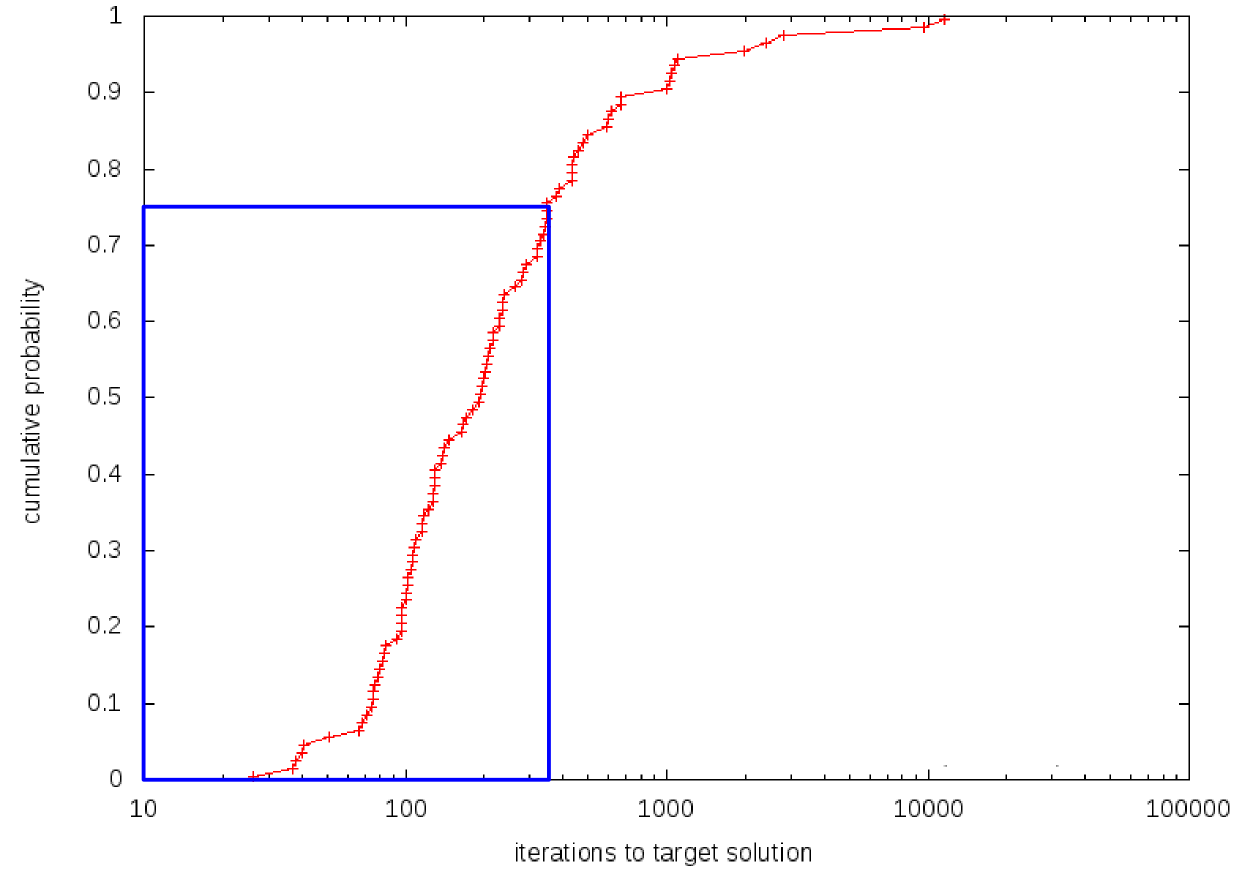
In most of the independent runs, the algorithm finds the target solution in relatively few iterations:



In most of the independent runs, the algorithm finds the target solution in relatively few iterations: 25% of the runs take fewer than 101 iterations

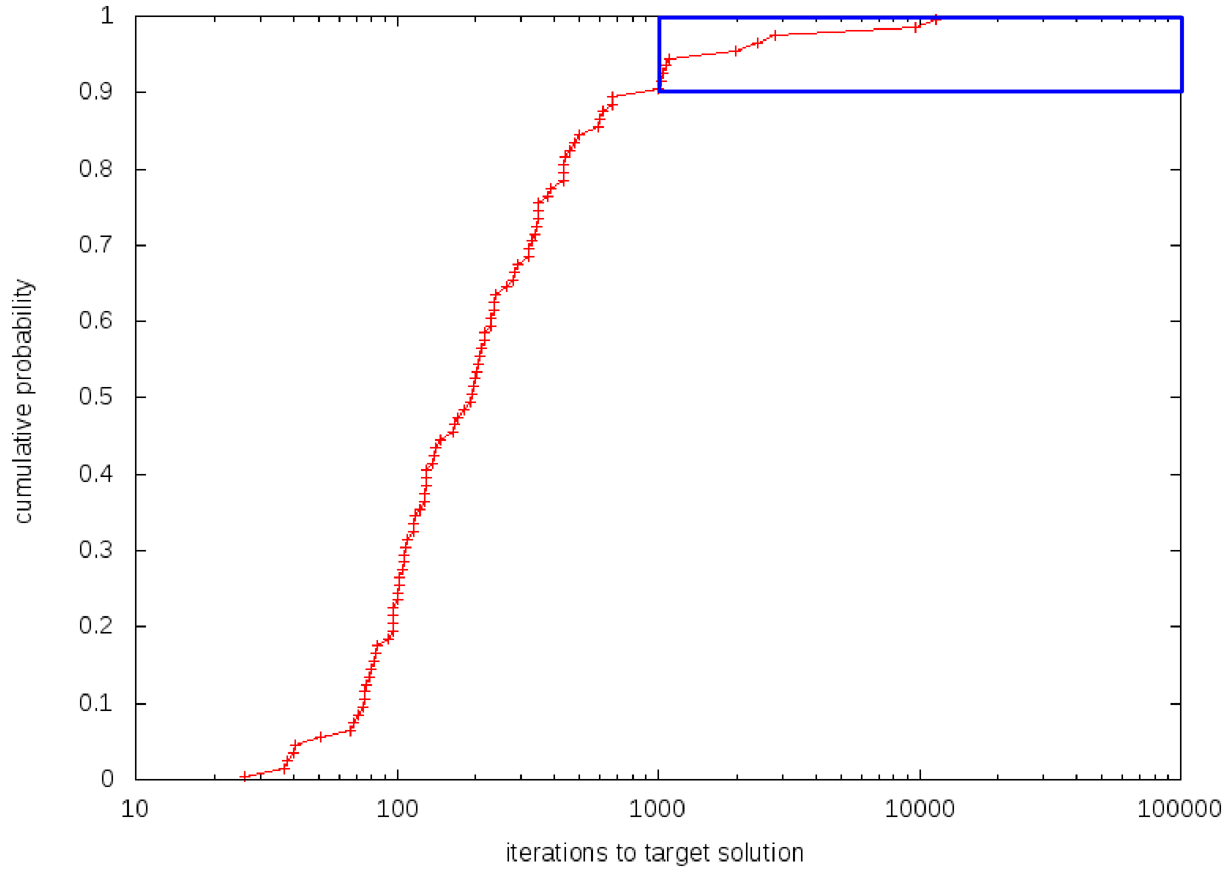


In most of the independent runs, the algorithm finds the target solution in relatively few iterations: 50% of the runs take fewer than 192 iterations

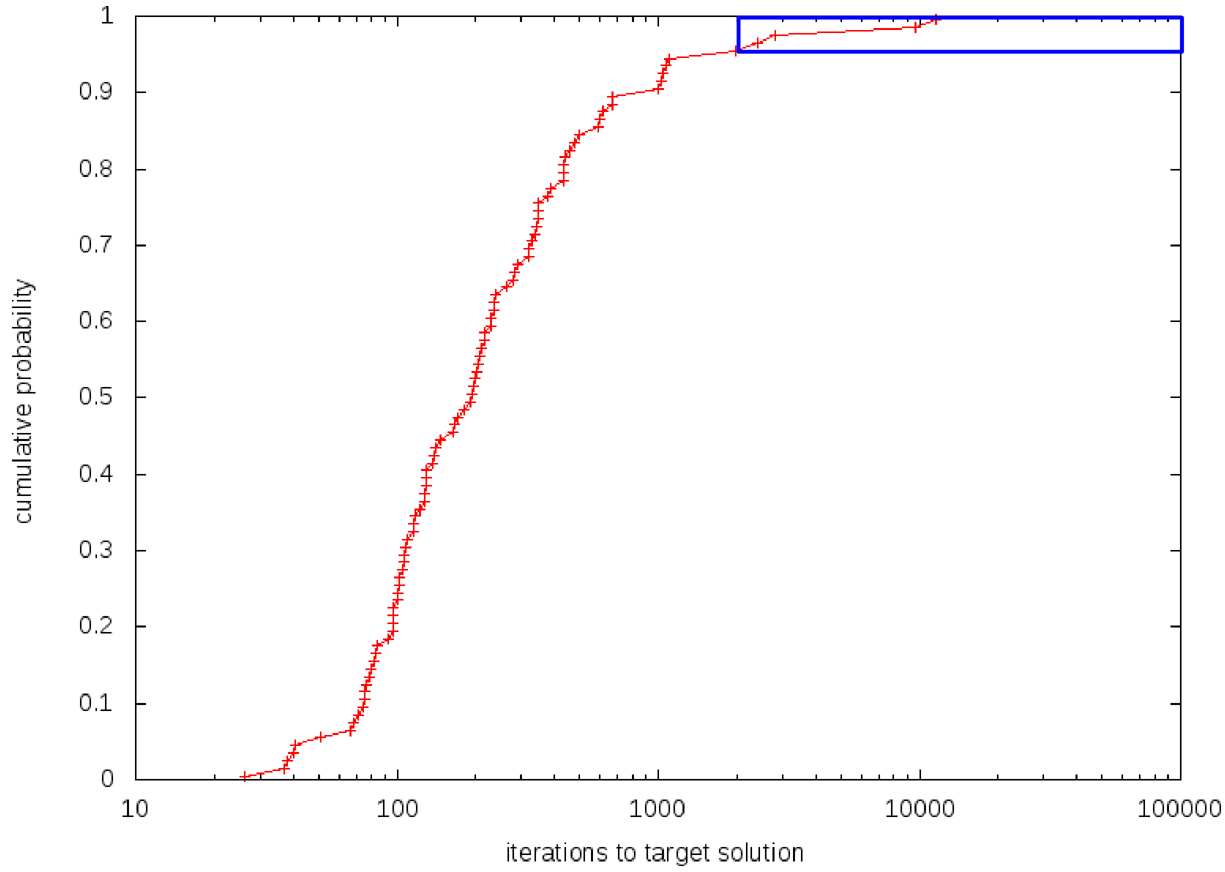


In most of the independent runs, the algorithm finds the target solution in relatively few iterations: 75% of the runs take fewer than 345 iterations

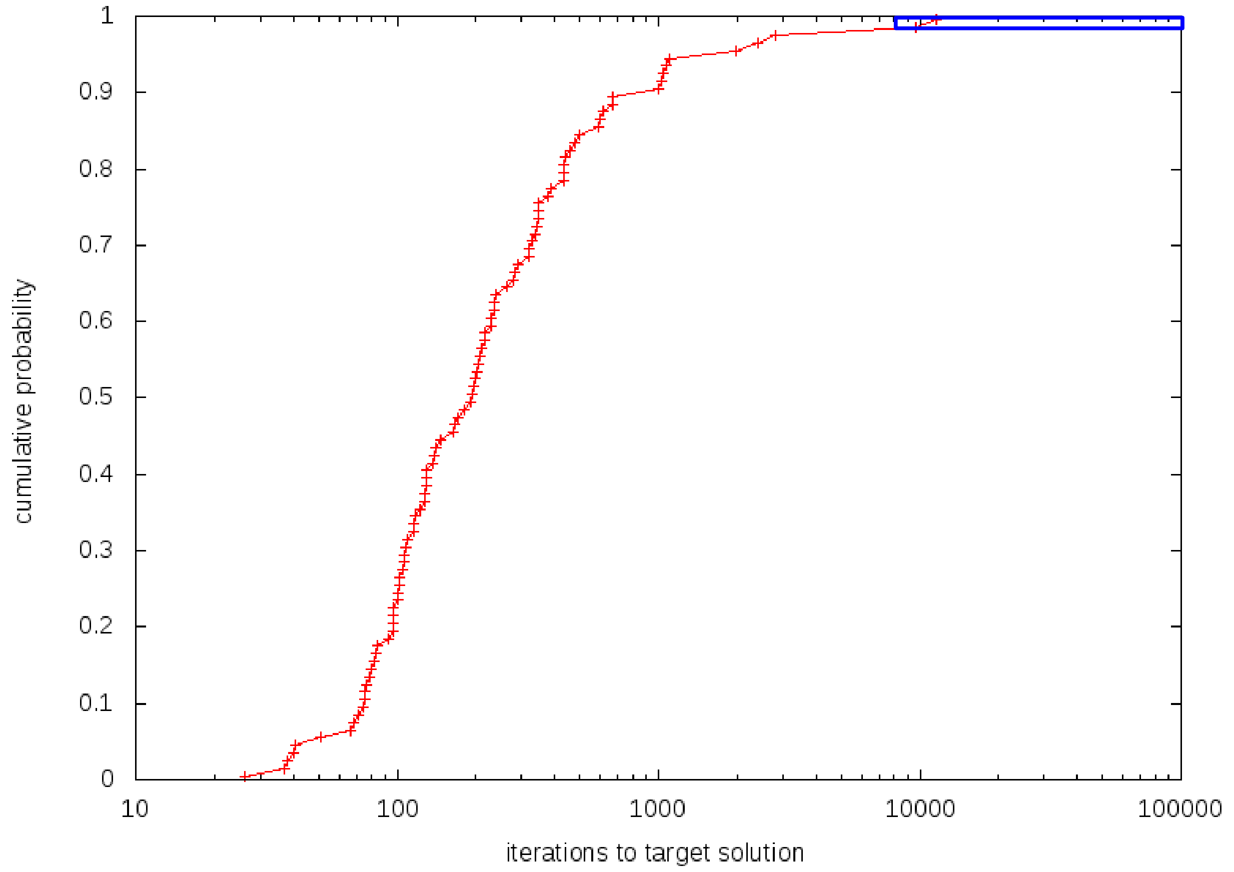
at&t Your world. Delivered.



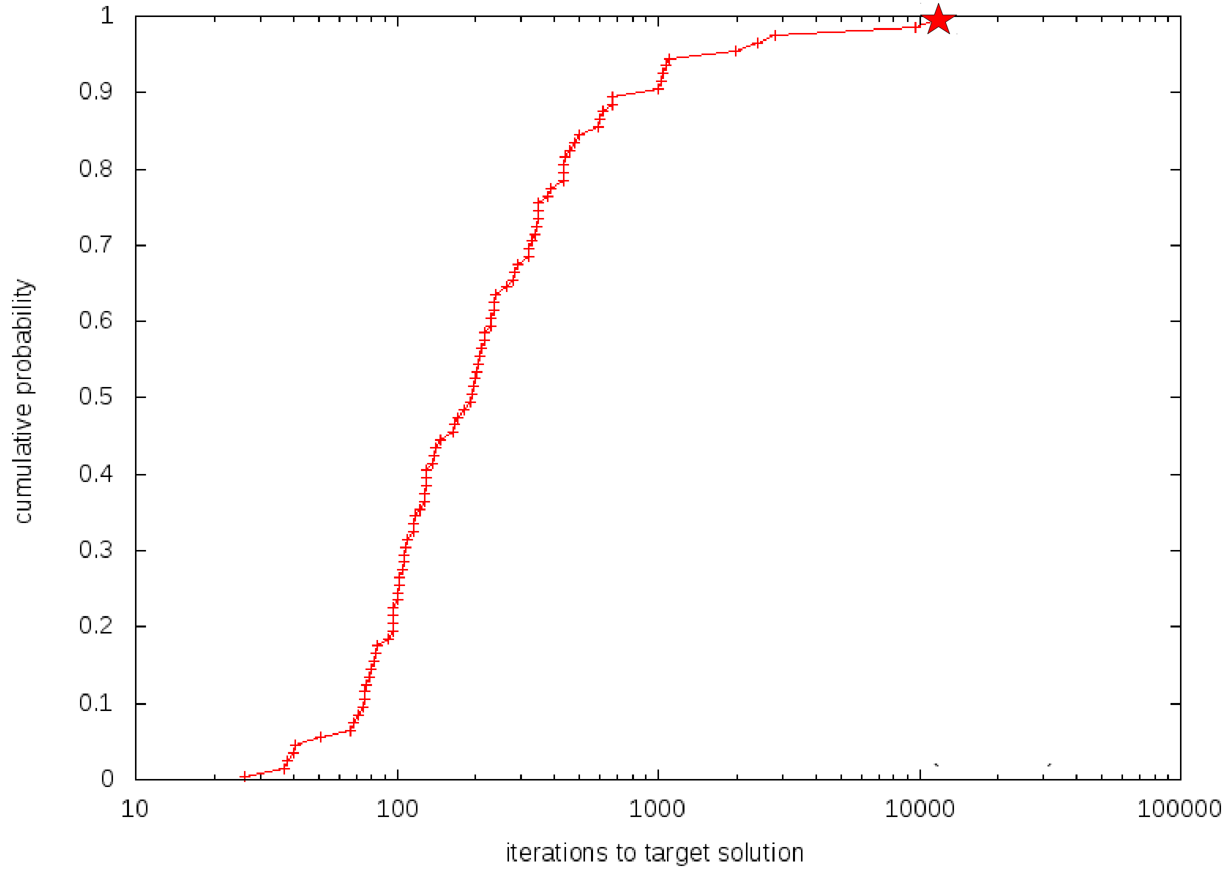
However, some runs take much longer: 10% of the runs take over 1000 iterations



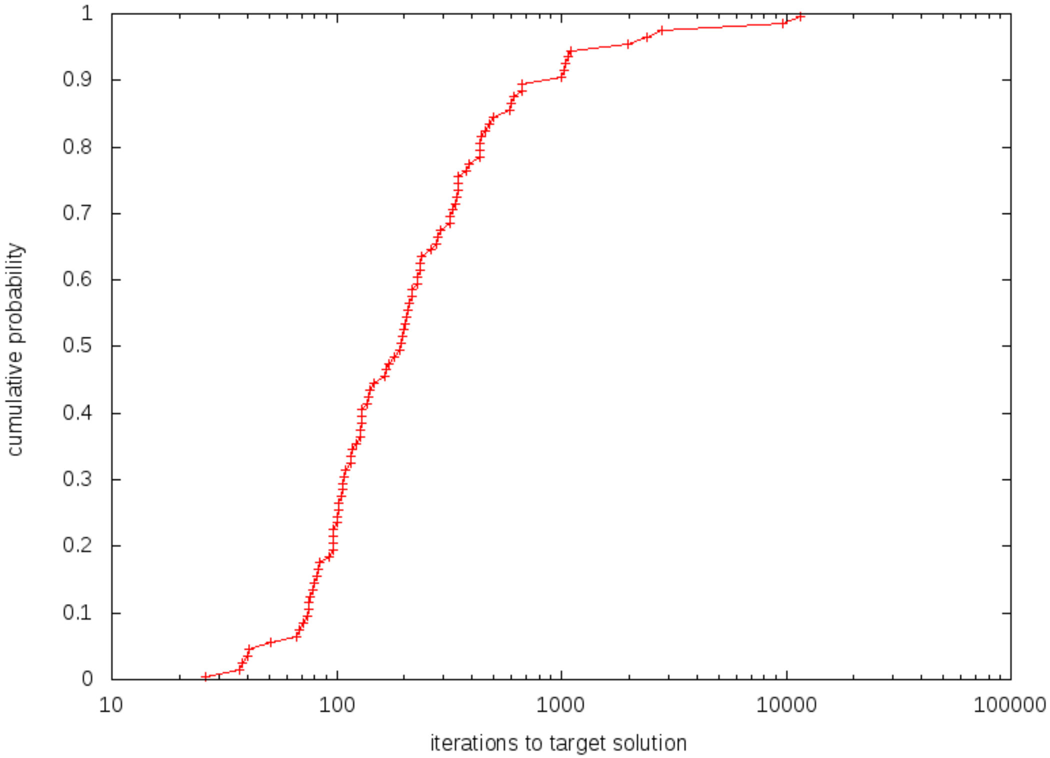
However, some runs take much longer: 5% of the runs take over 2000 iterations



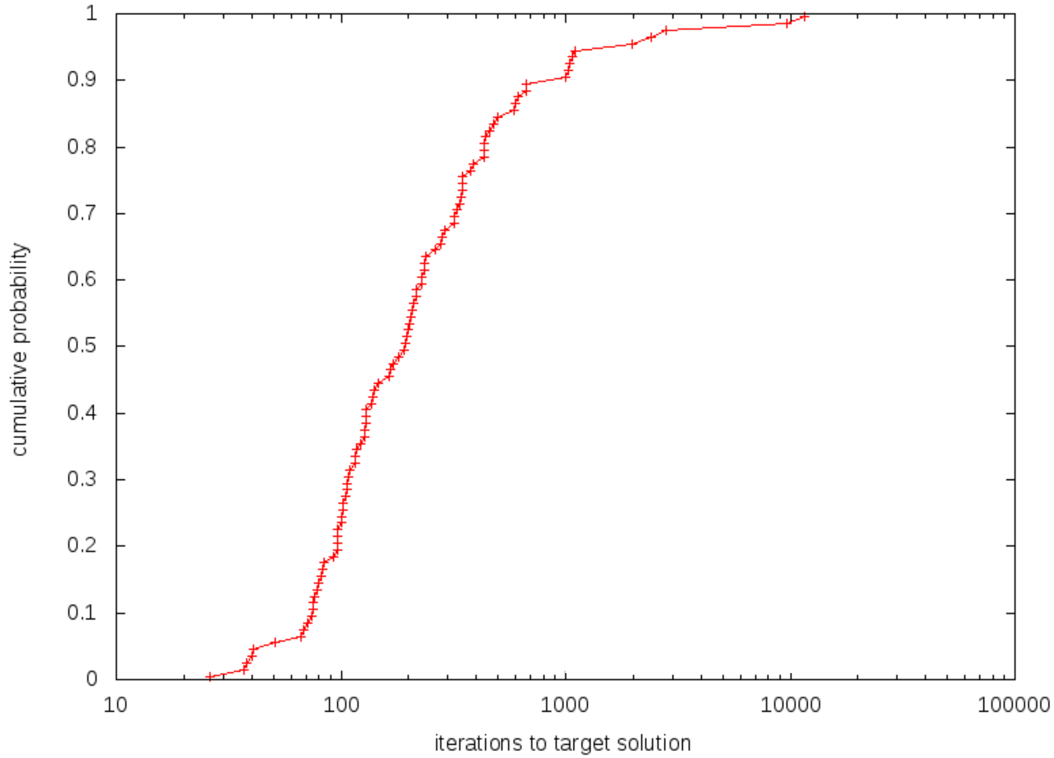
However, some runs take much longer: 2% of the runs take over 9715 iterations



However, some runs take much longer: the longest run took 11607 iterations



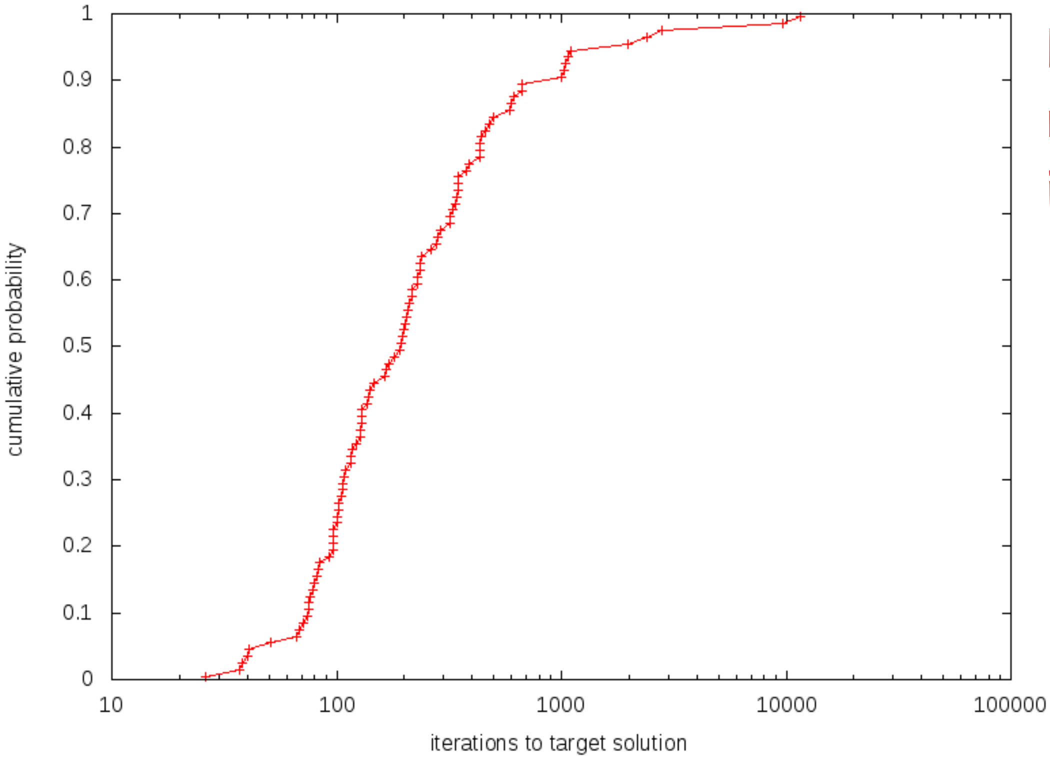
Probability that algorithm will take over 345 iterations: 25% = 1/4



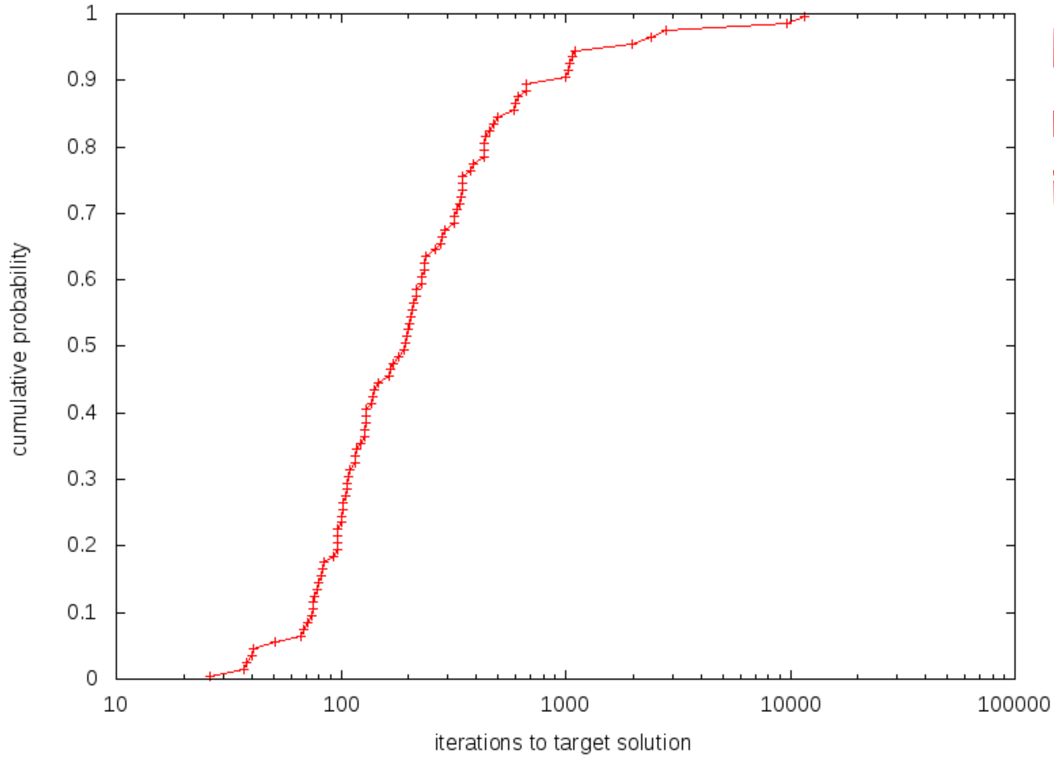
Probability that algorithm will take over 345 iterations: 25% = 1/4

By restarting algorithm after 345 iterations, probability that new run will take over 690 iterations: 25% = 1/4

Probability that algorithm with restart will take over 690 iterations: probability of taking over 345 X probability of taking over 690 iterations given it took over 345 = ¼ x ¼ = $1/4^2$

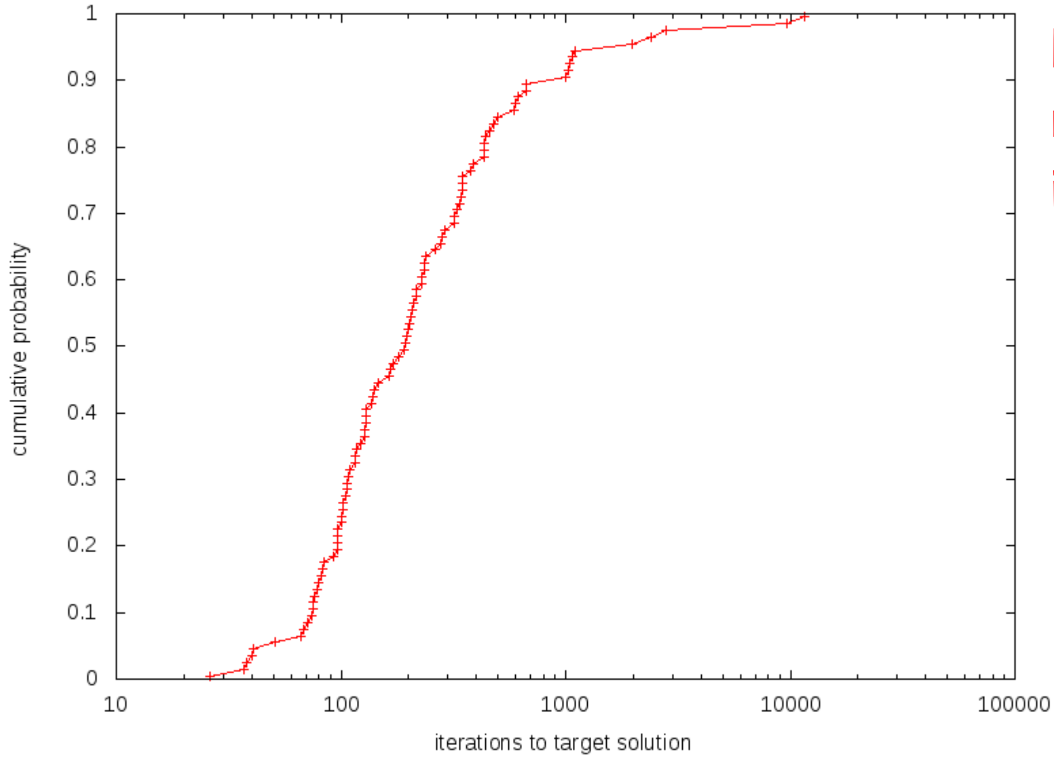


Probability that algorithm will still be running after K periods of 345 iterations: $1/4^K$



Probability that algorithm will still be running after K periods of 345 iterations: $1/4^K$

For example, probability that algorithm with restart will still be running after 1725 iterations (5 periods of 345 iterations): $1/4^5 \cong$ 0.0977%



Probability that algorithm will still be running after K periods of 345 iterations: $1/4^K$

For example, probability that algorithm with restart will still be running after 1725 iterations (5 periods of 345 iterations): $1/4^5 \cong$ 0.0977%

This is much less than the 5% probability that the algorithm without restart will take over 2000 iterations.

Restart strategies

- First proposed by Luby et al. (1993)
- They define a restart strategy as a finite sequence of time intervals $S = \{\tau_1, \tau_2, \tau_3, \ldots\}$ which define epochs τ_1, $\tau_1+\tau_2$, $\tau_1+\tau_2+\tau_3$, ... when the algorithm is restarted from scratch.
- Luby et al. (1993) prove that the optimal restart strategy uses $\tau_1 = \tau_2 = \tau_3 = \cdots = \tau^*$, where τ^* is a constant.

Restart strategies

- Luby et al. (1993)
- Kautz et al. (2002)
- Palubeckis (2004)
- Sergienko et al. (2004)
- Nowicki & Smutnicki (2005)
- D'Apuzzo et al. (2006)
- Shylo et al. (2011a)
- Shylo et al. (2011b)
- Resende & Ribeiro (2011)

Restart strategy for BRKGA

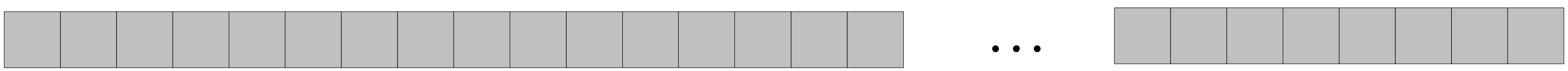
- Recall the restart strategy of Luby et al. where equal time intervals $\tau_1 = \tau_2 = \tau_3 = \cdots = \tau^*$ pass between restarts.
- Strategy requires τ^* as input.
- Since we have no prior information as to the runtime distribution of the heuristic, we run the risk of:
  - choosing τ^* too small: restart variant may take long to converge
  - choosing τ^* too big: restart variant may become like no-restart variant

Restart strategy for BRKGA

- We conjecture that number of iterations between improvement of the incumbent (best so far) solution varies less w.r.t. heuristic/ instance/ target than run times.
- We propose the following restart strategy: Keep track of the last generation when the incumbent improved and restart BRKGA if K generations have gone by without improvement.
- We call this strategy restart(K)

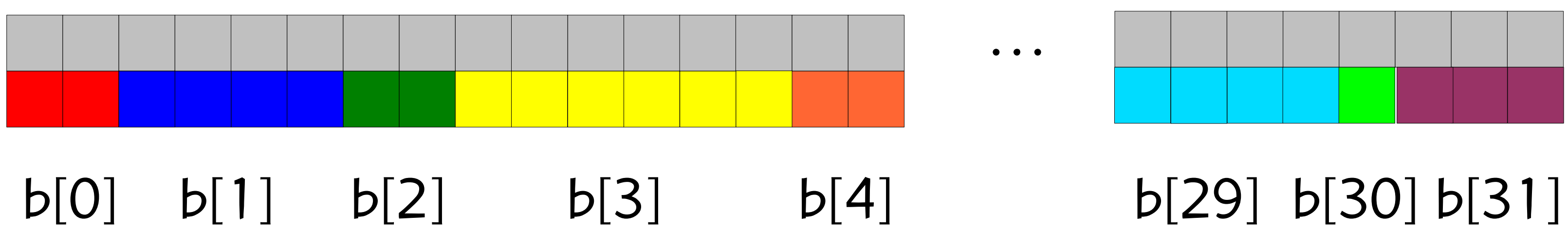
Example of restart strategy for BRKGA: Load balancing

Given an ordered sequence of 1024 integers p[0], p[1], ..., p[1023]



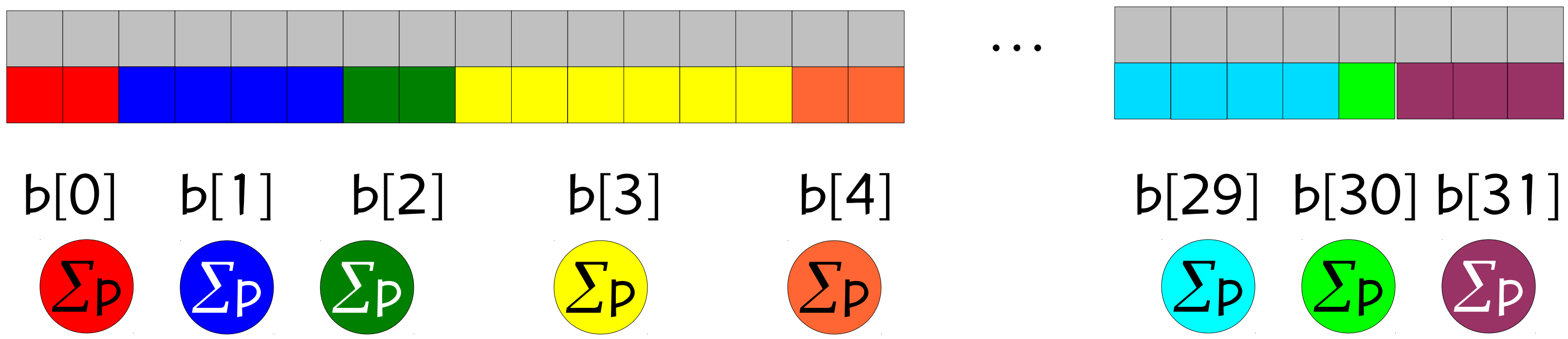
Example of restart strategy for BRKGA: Load balancing

Place consecutive numbers in 32 buckets b[0], b[1], ..., b[31]



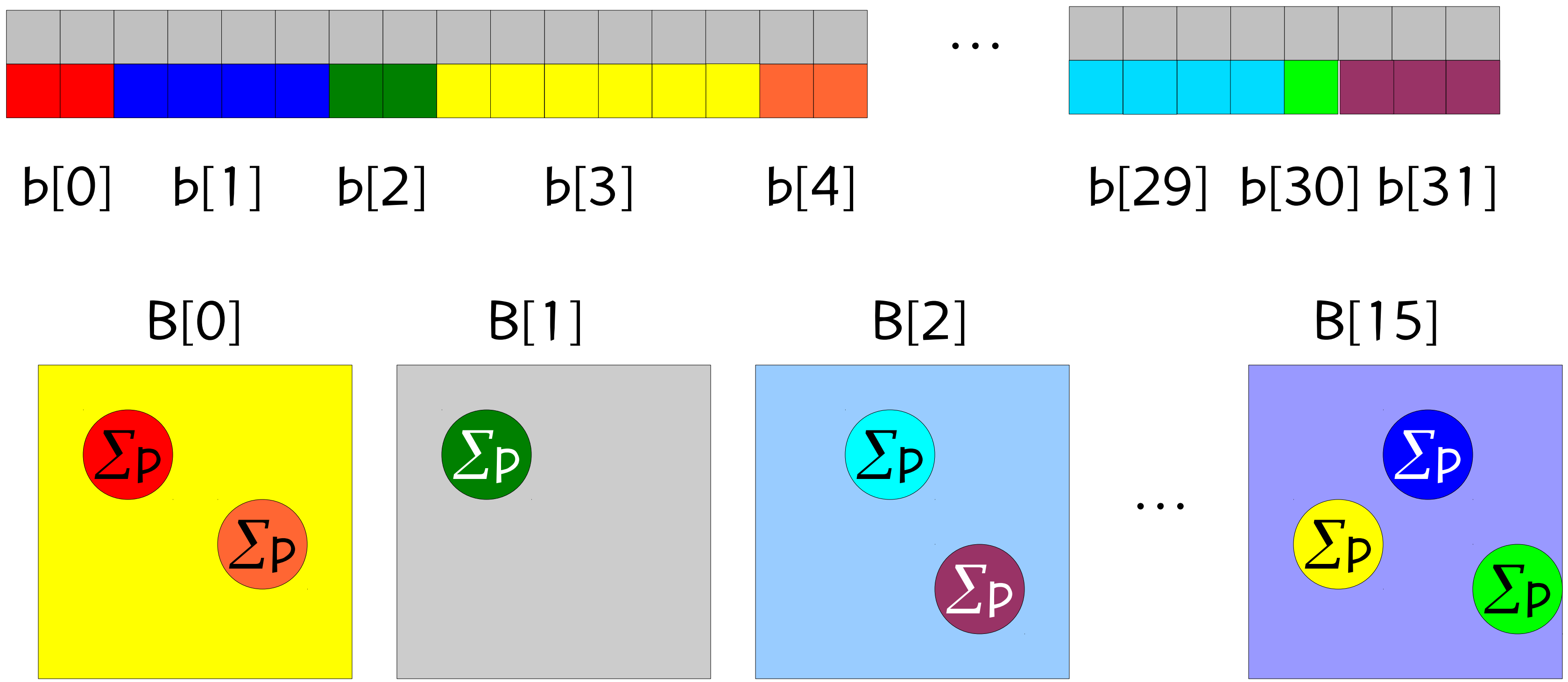
Example of restart strategy for BRKGA: Load balancing

Add the numbers in each bucket b[0], b[1], ..., b[31]



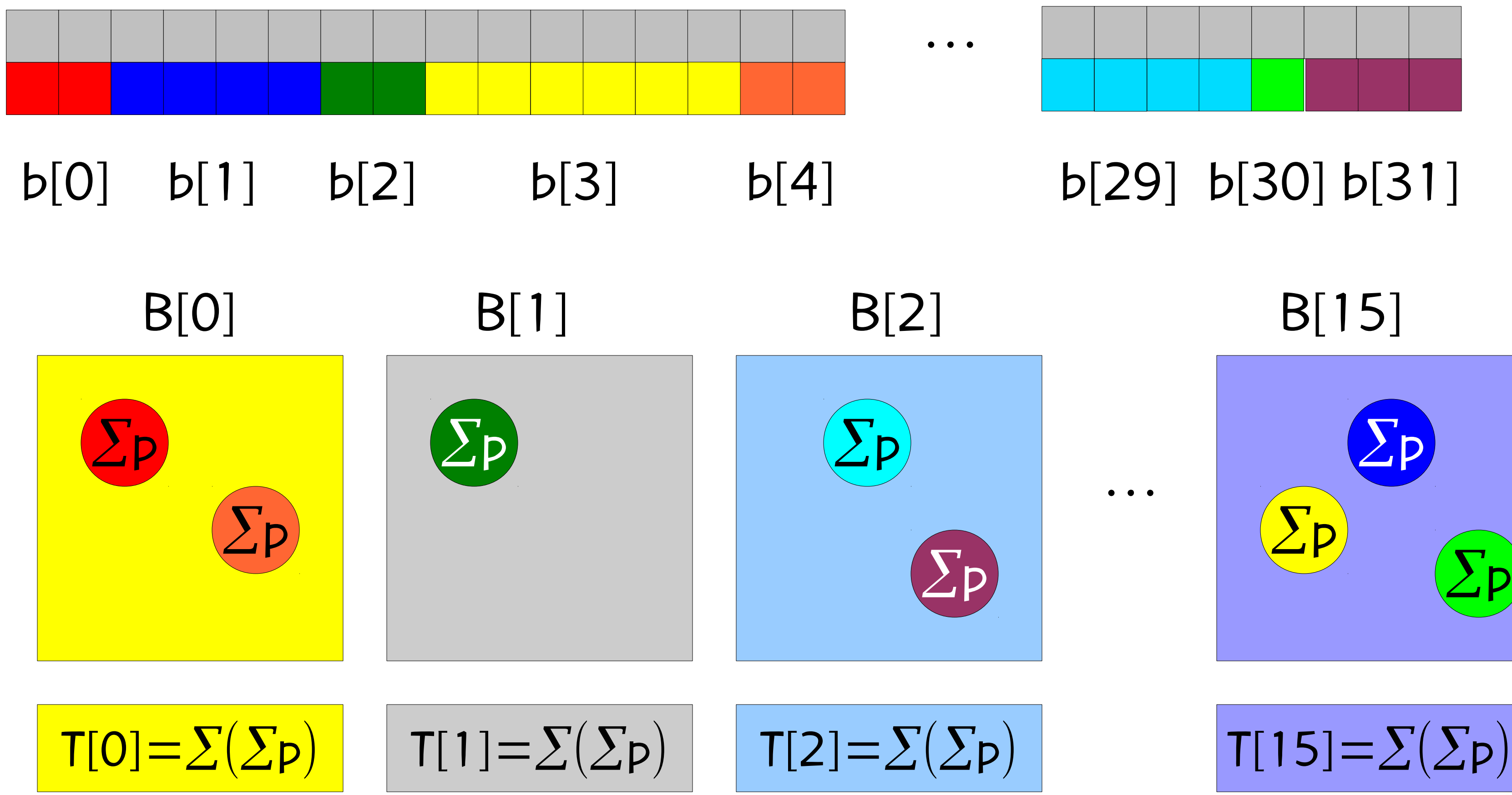
Example of restart strategy for BRKGA: Load balancing

Place the buckets in 16 bins B[0], B[1], ..., B[15]



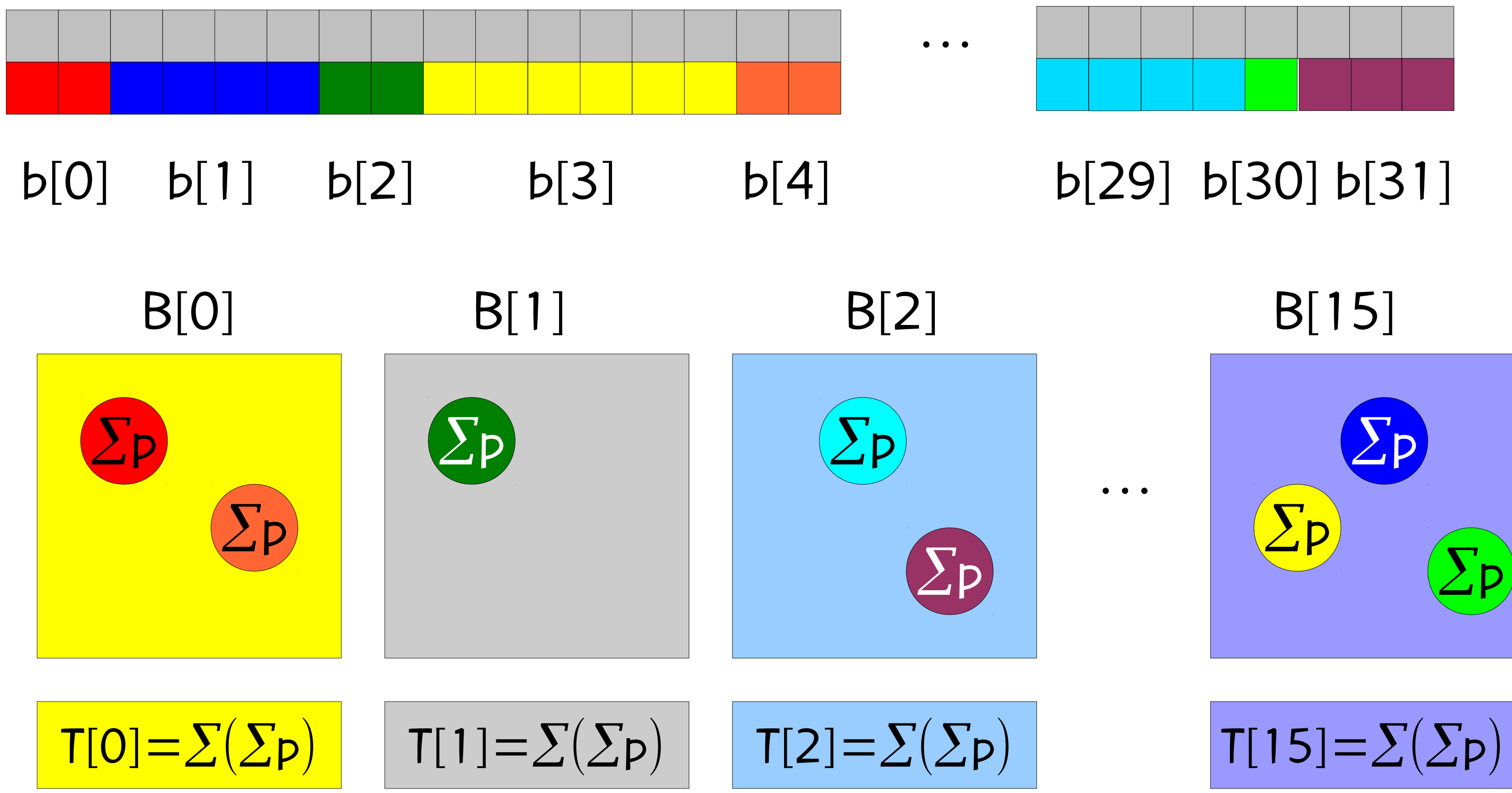
Example of restart strategy for BRKGA: Load balancing

Add up the numbers in each bin B[0], B[1], ..., B[15]



Example of restart strategy for BRKGA: Load balancing

OBJECTIVE: Minimize { Maximum (T[0], T[1], ..., T[15]) }



Example of restart strategy for BRKGA: Load balancing

Encoding

X = [x[1], x[2], ..., x[32] | x[32+1], x[32+2], ..., x[32+16]]

Decoding

x[1], x[2], ..., x[32] are used to define break points for buckets

x[32+1], x[32+2], ..., x[32+16] are used to determine to which bins the buckets are assigned

Example of restart strategy for BRKGA: Load balancing

Encoding

X = [x[1], x[2], ..., x[32] | x[32+1], x[32+2], ..., x[32+16]]

Decoding

x[1], x[2], ..., x[32] are used to define break points for buckets

Size of bucket i = floor (1024 × x[i]/(x[1]+x[2]+···+x[32])), i=1,...,15

Size of bucket 16 = 1024 – sum of sizes of first 15 buckets

Example of restart strategy for BRKGA: Load balancing

Encoding

X = [x[1], x[2], ..., x[32] | x[32+1], x[32+2], ..., x[32+16]]

Decoding

x[1], x[2], ..., x[32] are used to define break points for buckets

x[32+1], x[32+2], ..., x[32+16] are used to determine to which bins the buckets are assigned

Bin that bucket i is assigned to = floor (16 × x[32+i]) + 1

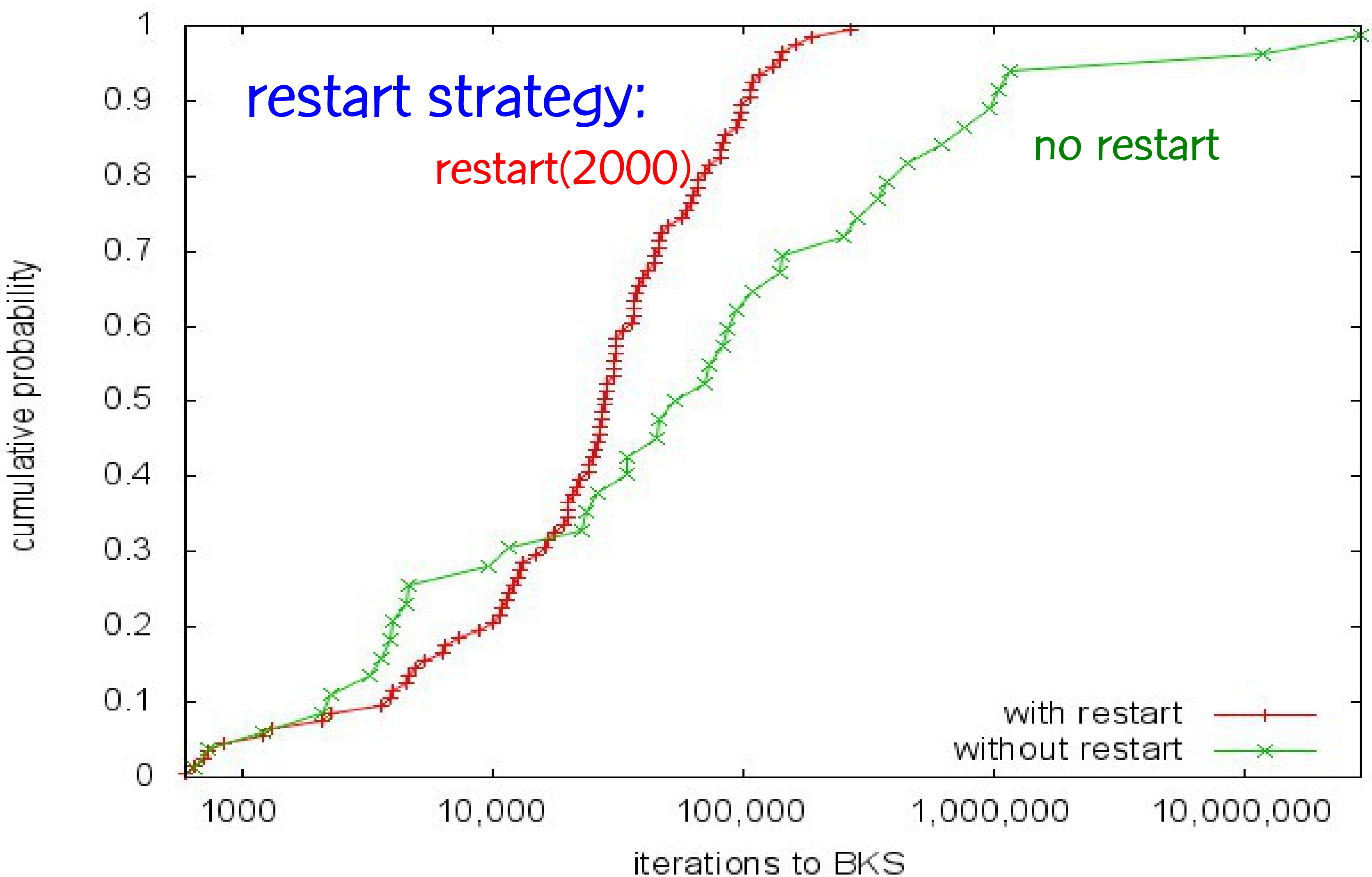
Example of restart strategy for BRKGA: Load balancing

Decoding (Local search phase)

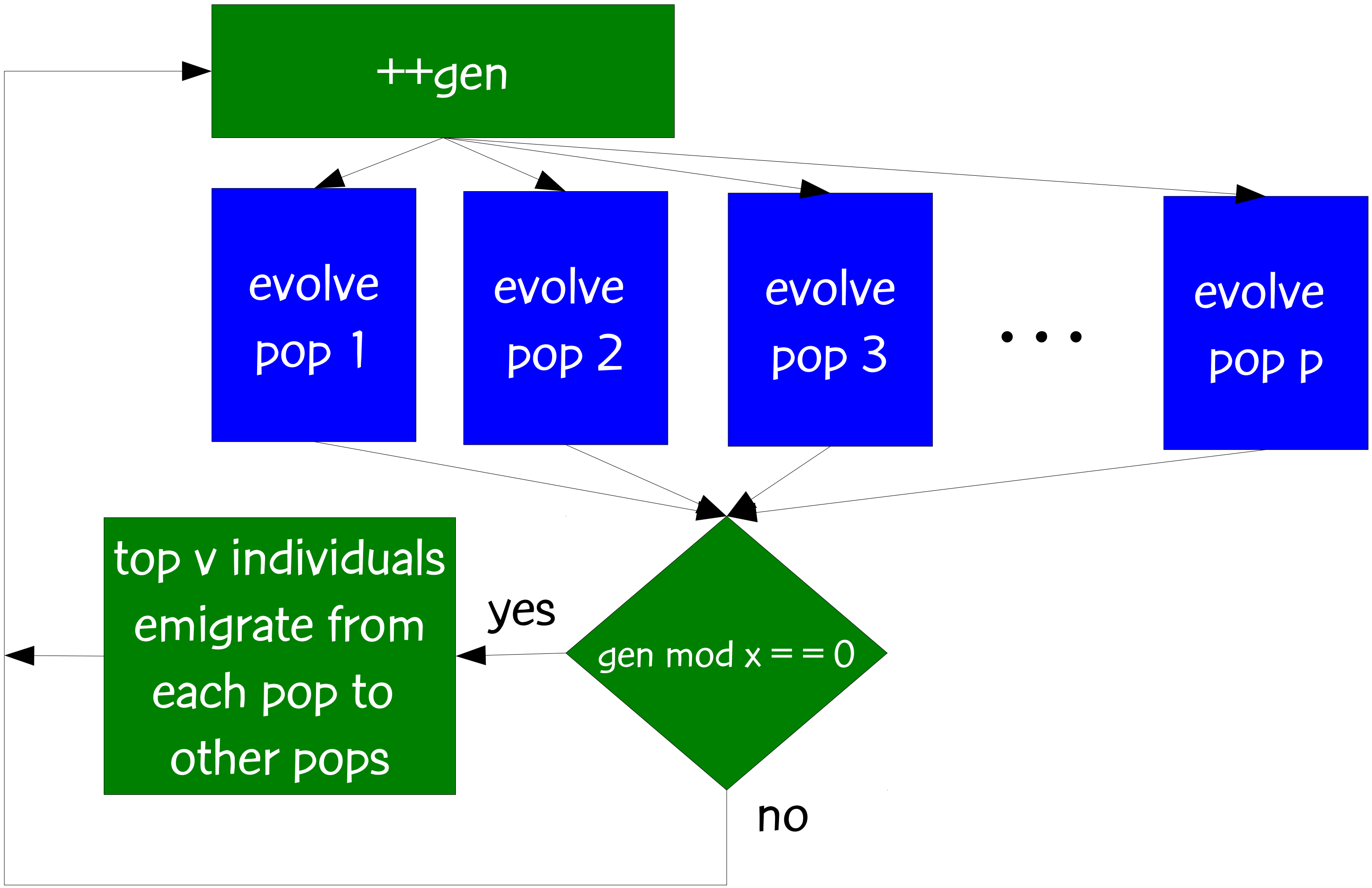
- **while** (there exists a bucket in the most loaded bin that can be moved to another bin and not increase the maximum load) **then**
  - move that bucket to that bin
- **end while**

Make necessary chromosome adjustments to last 16 random keys of vector of random keys to reflect changes made in local search phase: Add or subtract an integer value from chromosome of bucket that moved to new bin.

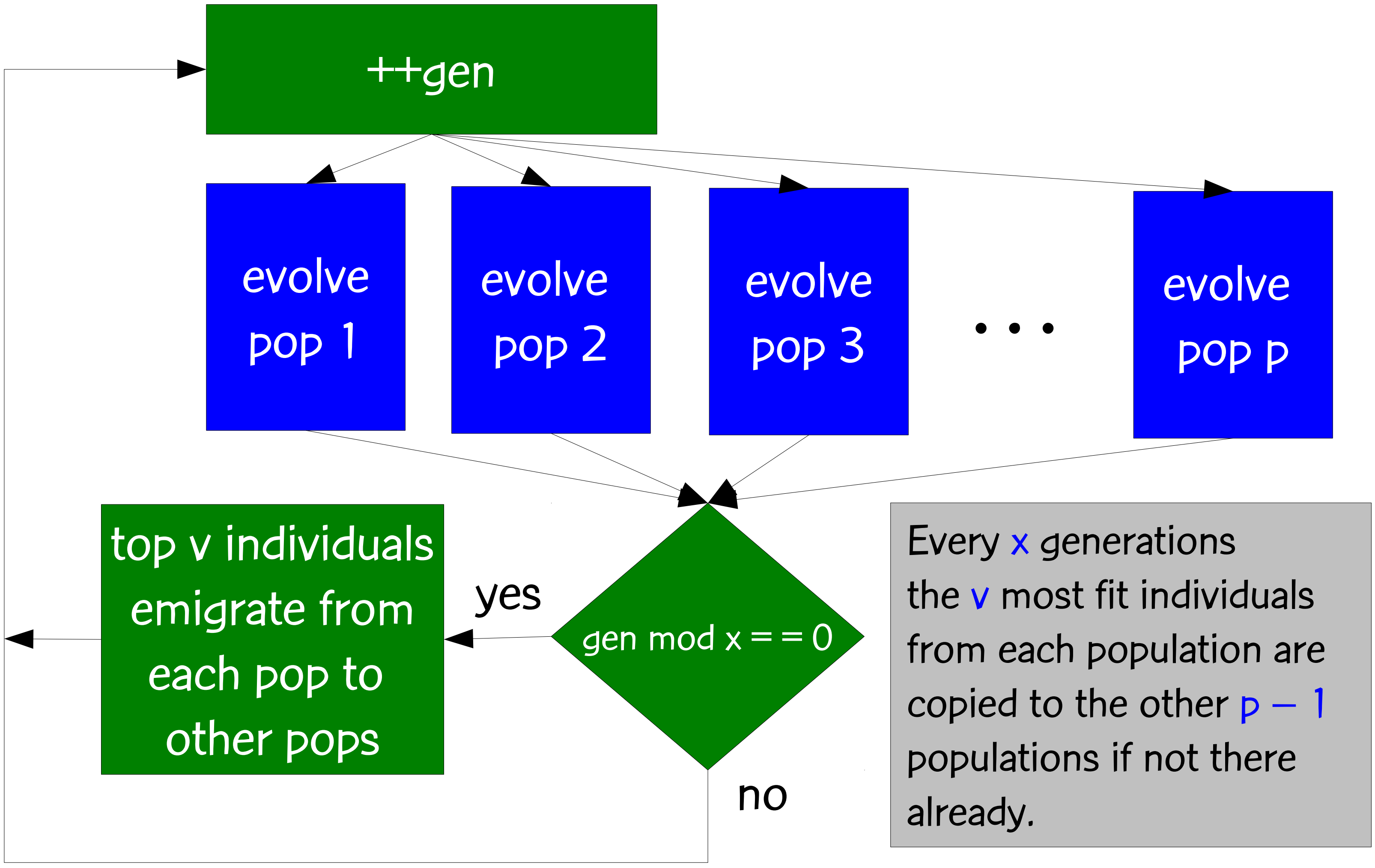
Example of restart strategy for BRKGA: Load balancing



BRKGA with p parallel populations



BRKGA with p parallel populations



Initialize population with some non-random individuals

It is often useful to initialize the first population with some individuals not generated totally at random.

- Generate some individuals using simple heuristics, e.g. Buriol, M.G.C.R., Ribeiro, & Thorup (2005)
- Formulate 0-1 integer program and solve linear programming (LP) relaxation and use LP solution as individual, e.g. Andrade, Miyazawa, M.G.C.R., & Toso (2012)

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

Specifying a biased random-key GA

Parameters:

- Size of population
- Parallel population parameters
- Size of elite partition
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters
- Size of elite partition
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, $p = 3$, $v = 2$, and $x = 200$
- Size of elite partition
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, $p = 3$, $v = 2$, and $x = 200$
- Size of elite partition: 15-25% of population
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, $p = 3$, $v = 2$, and $x = 200$
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, p = 3 , v = 2, and x = 200
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability: > 0.5, say 0.7
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, p = 3 , v = 2, and x = 200
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability: > 0.5, say 0.7
- Restart strategy parameter: a function of N, say 2N or 10N
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, p = 3 , v = 2, and x = 200
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability: > 0.5, say 0.7
- Restart strategy parameter: a function of N, say 2N or 10N
- Stopping criterion: e.g. time, # generations, solution quality, # generations without improvement

brkgaAPI: A C++ API for BRKGA

- Efficient and easy-to-use object oriented application programming interface (API) for the algorithmic framework of BRKGA.

brkgaAPI: A C++ API for BRKGA

- Efficient and easy-to-use object oriented application programming interface (API) for the algorithmic framework of BRKGA.
- Cross-platform library handles large portion of problem independent modules that make up the framework, e.g.
  - population management
  - evolutionary dynamics

brkgaAPI: A C++ API for BRKGA

- Efficient and easy-to-use object oriented application programming interface (API) for the algorithmic framework of BRKGA.
- Cross-platform library handles large portion of problem independent modules that make up the framework, e.g.
  - population management
  - evolutionary dynamics
- Implemented in C++ and may benefit from shared-memory parallelism if available.

brkgaAPI: A C++ API for BRKGA

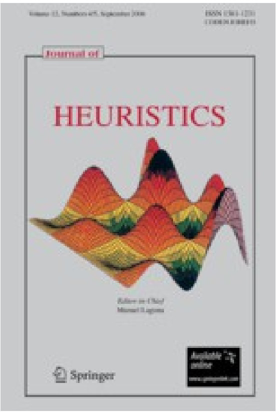
- Efficient and easy-to-use object oriented application programming interface (API) for the algorithmic framework of BRKGA.
- Cross-platform library handles large portion of problem independent modules that make up the framework, e.g.
  - population management
  - evolutionary dynamics
- Implemented in C++ and may benefit from shared-memory parallelism if available.
- User only needs to implement problem-dependent decoder.

brkgaAPI: A C++ API for BRKGA

Paper: Rodrigo F. Toso and M.G.C.R., "A C++ Application Programming Interface for Biased Random-Key Genetic Algorithms," AT&T Labs Technical Report, Florham Park, August 2011.

Software: http://www.research.att.com/~mgcr/src/brkgaAPI

Reference



J.F. Gonçalves and M.G.C.R., "Biased random-key genetic algorithms for combinatorial optimization," J. of Heuristics, vol.17, pp. 487-525, 2011.

Tech report version:

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

Reference

M.G.C.R., "Biased random-key genetic algorithms with applications in telecommunications," TOP, vol. 20, pp. 120-153, 2012.

Tech report version:

http://www.research.att.com/~mgcr/doc/brkga-telecom.pdf

Thanks!

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

http://www.research.att.com/~mgcr

Thanks!

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

http://www.research.att.com/~mgcr

See you tomorrow for some applications of BRKGA.

Biased random-key genetic algorithms: A tutorial

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

mgcr@research.att.com

Tutorial given at CLAIO/SBPO 2012
Rio de Janeiro, Brazil ✤ September 2012

AT&T Shannon Laboratory
Florham Park, New Jersey

Part 2 of tutorial

Summary: Day 1

- Basic concepts of combinatorial and continuous global optimization
- Basic concepts of genetic algorithms
- Random-key genetic algorithm of Bean (1994)
- Biased random-key genetic algorithms (BRKGA)
  - Encoding / Decoding
  - Initial population
  - Evolutionary mechanisms
  - Problem independent / problem dependent components
  - Multi-start strategy
  - Restart strategy
  - Multi-population strategy
  - Specifying a BRKGA
- Application programming interface (API) for BRKGA

Summary: Day 2

- Applications of BRKGA
  - Set covering
  - Packing rectangles
  - Packet routing on the Internet
  - Handover minimization in mobility networks
  - Continuous global optimization
- Overview of literature & concluding remarks

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

Specifying a biased random-key GA

Parameters:

- Size of population
- Parallel population parameters
- Size of elite partition
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters
- Size of elite partition
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, $p = 3$, $v = 2$, and $x = 200$
- Size of elite partition
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, p = 3 , v = 2, and x = 200
- Size of elite partition: 15-25% of population
- Size of mutant set
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, p = 3 , v = 2, and x = 200
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, p = 3 , v = 2, and x = 200
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability: > 0.5, say 0.7
- Restart strategy parameter
- Stopping criterion

Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, p = 3 , v = 2, and x = 200
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability: > 0.5, say 0.7
- Restart strategy parameter: a function of N, say 2N or 10N
- Stopping criterion

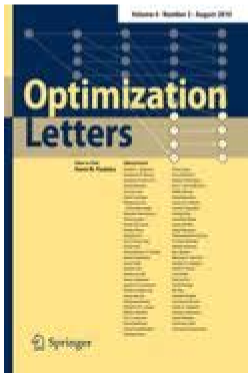
Specifying a biased random-key GA

Parameters:

- Size of population: a function of N, say N or 2N
- Parallel population parameters: say, $p = 3$, $v = 2$, and $x = 200$
- Size of elite partition: 15-25% of population
- Size of mutant set: 5-15% of population
- Child inheritance probability: > 0.5, say 0.7
- Restart strategy parameter: a function of N, say 2N or 10N
- Stopping criterion: e.g. time, # generations, solution quality, # generations without improvement

Some applications of BRKGA

Steiner triple covering



M.G.C.R., 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, vol. 6, pp. 605-619, 2012.

tech report: http://www.research.att.com/~mgcr/doc/brkga-stn.pdf

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	?	M.G.C.R. et al. (2012)
stn729	729	88452	617	?	M.G.C.R. et al. (2012)

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

Let $Y = (Y_1, Y_2, \ldots, Y_n)$ 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, ..., 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 adjustment

Implementation issues

Chromosome adjustment: 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 $X_j \geq ½$ and $j \in J^*$ or if $X_j < ½$ and $j \notin J^*$

X_j changes to $1 - X_j$ if $X_j < ½$ and $j \in J^*$ or if $X_j \geq ½$ and $j \notin J^*$

at&t Your world. Delivered.

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

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 replace 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 replace 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.

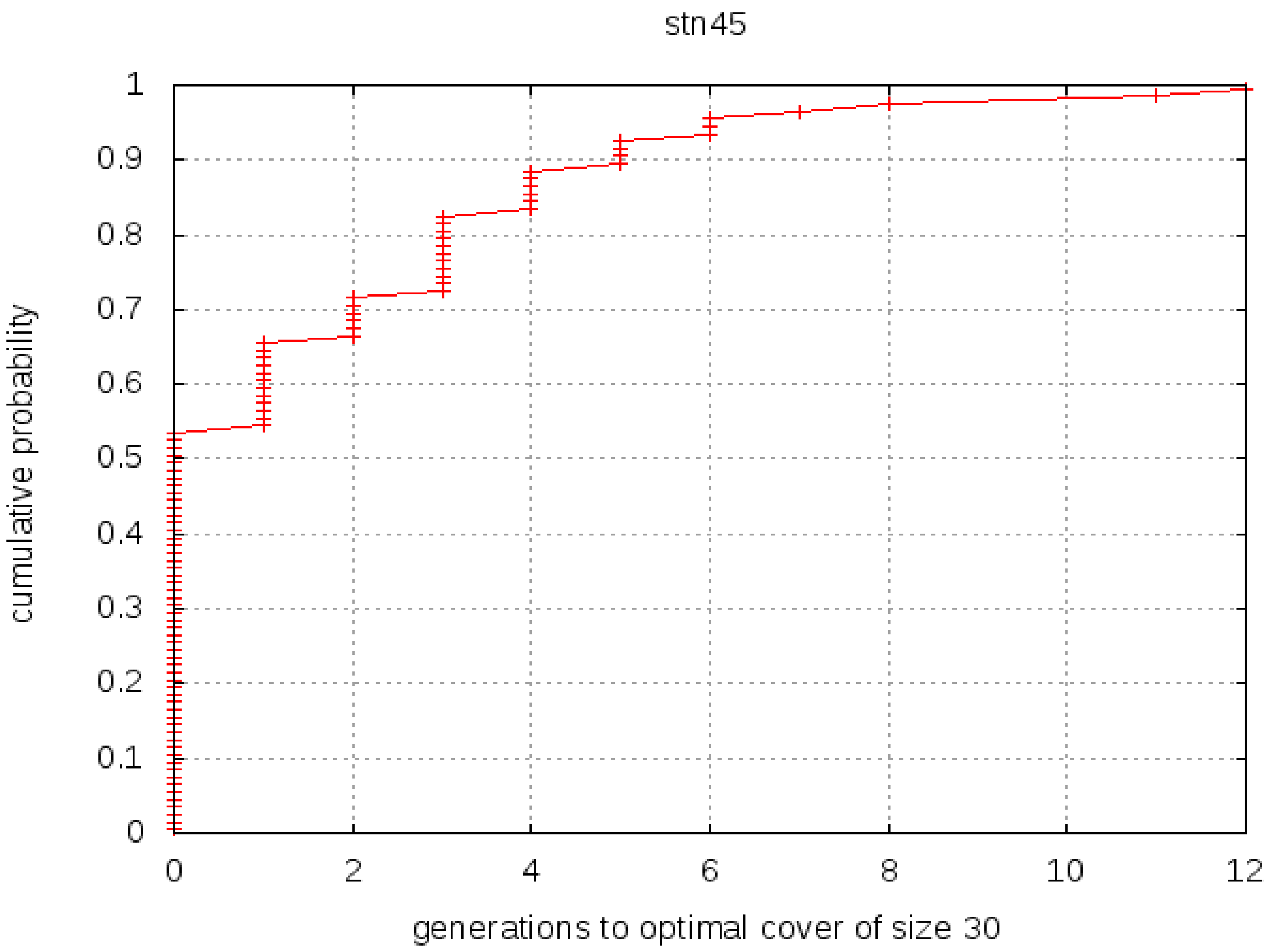
Experiments on instances with known optimal covers

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

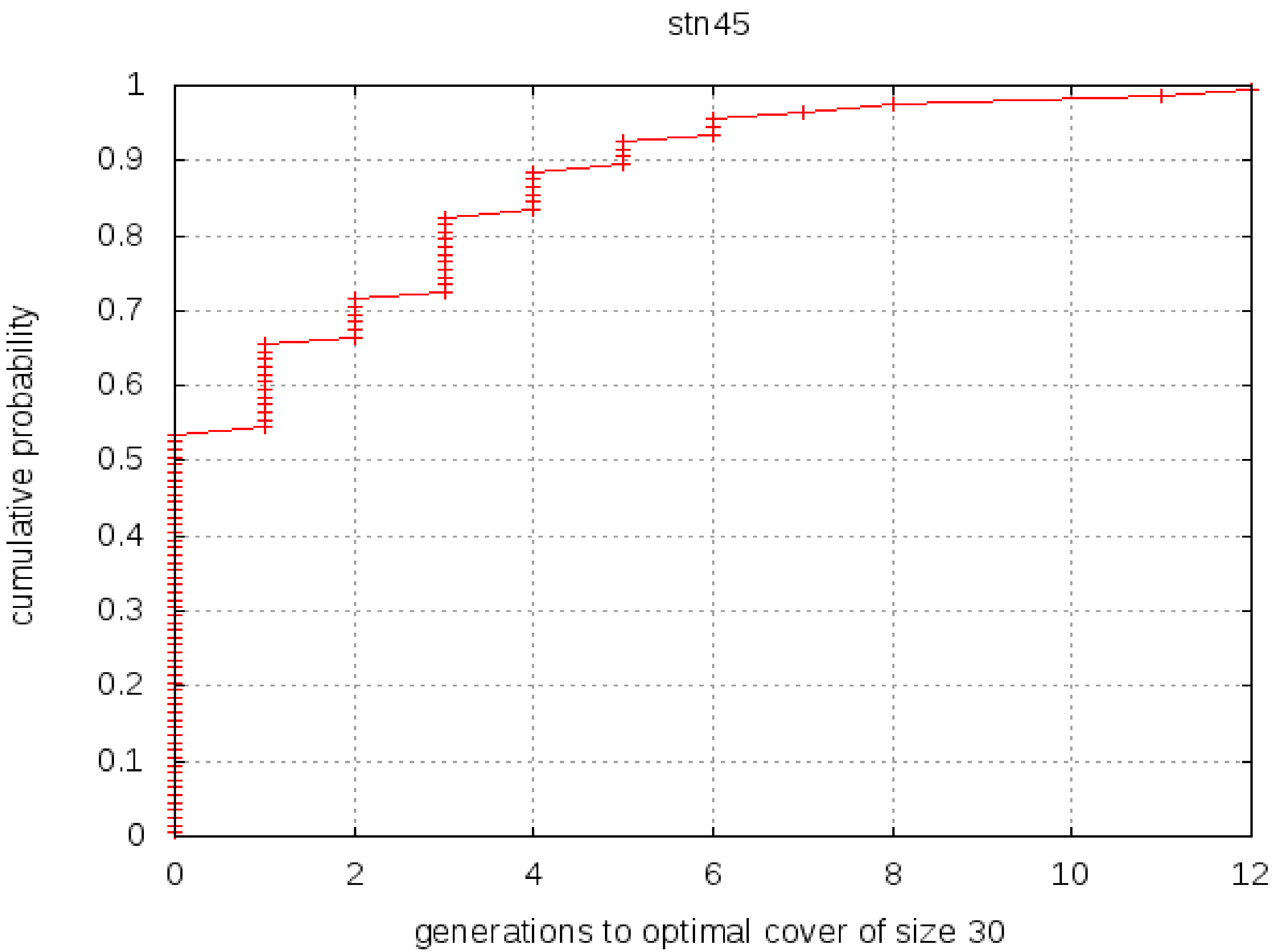
On all 100 runs for each instance, the algorithm found an optimal cover.

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

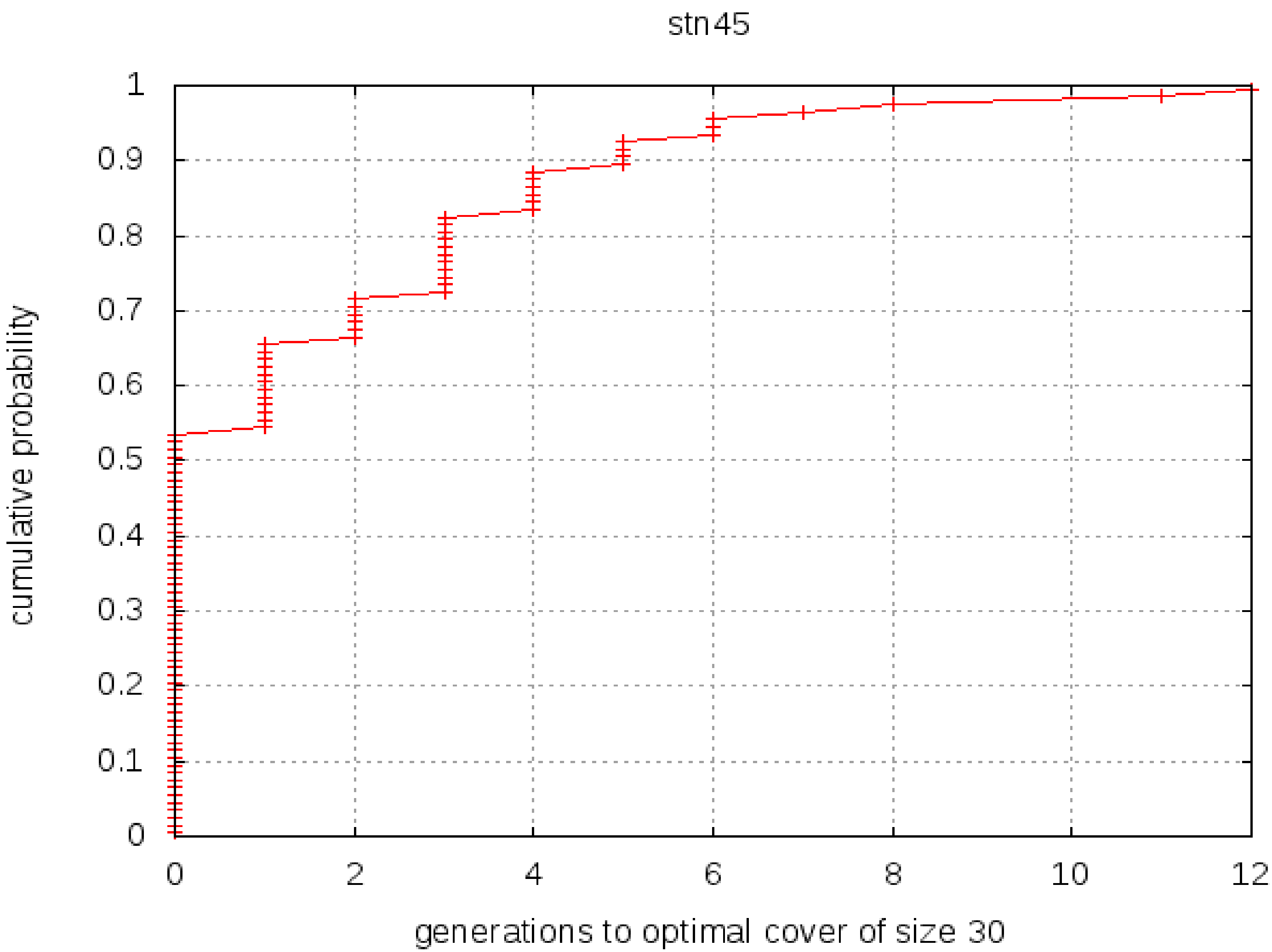
On stn81 an optimal cover was found in the initial population in 99 of the 100 runs. In the remaining run, an optimal cover was found in the second iteration.



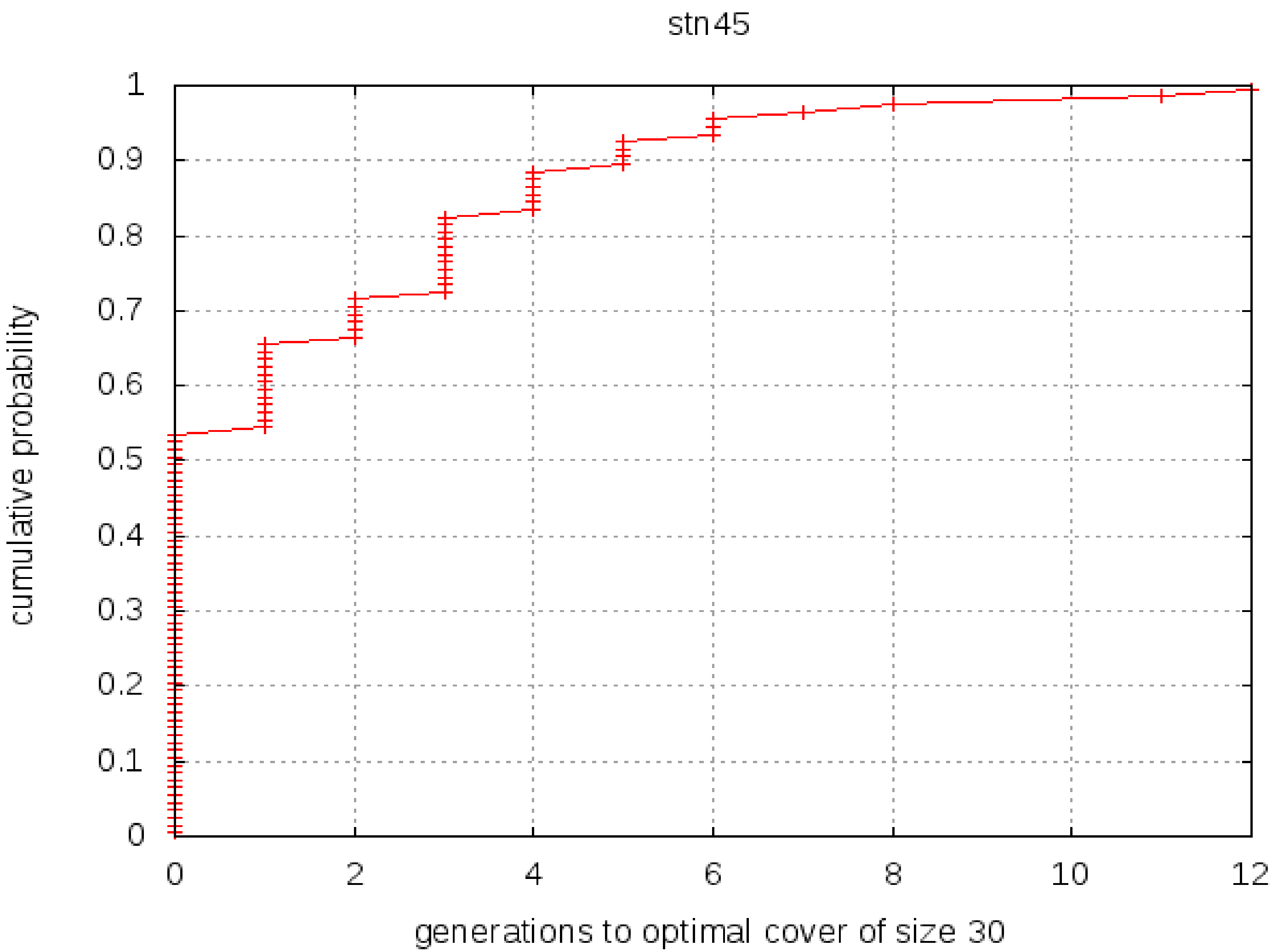
Instance stn45



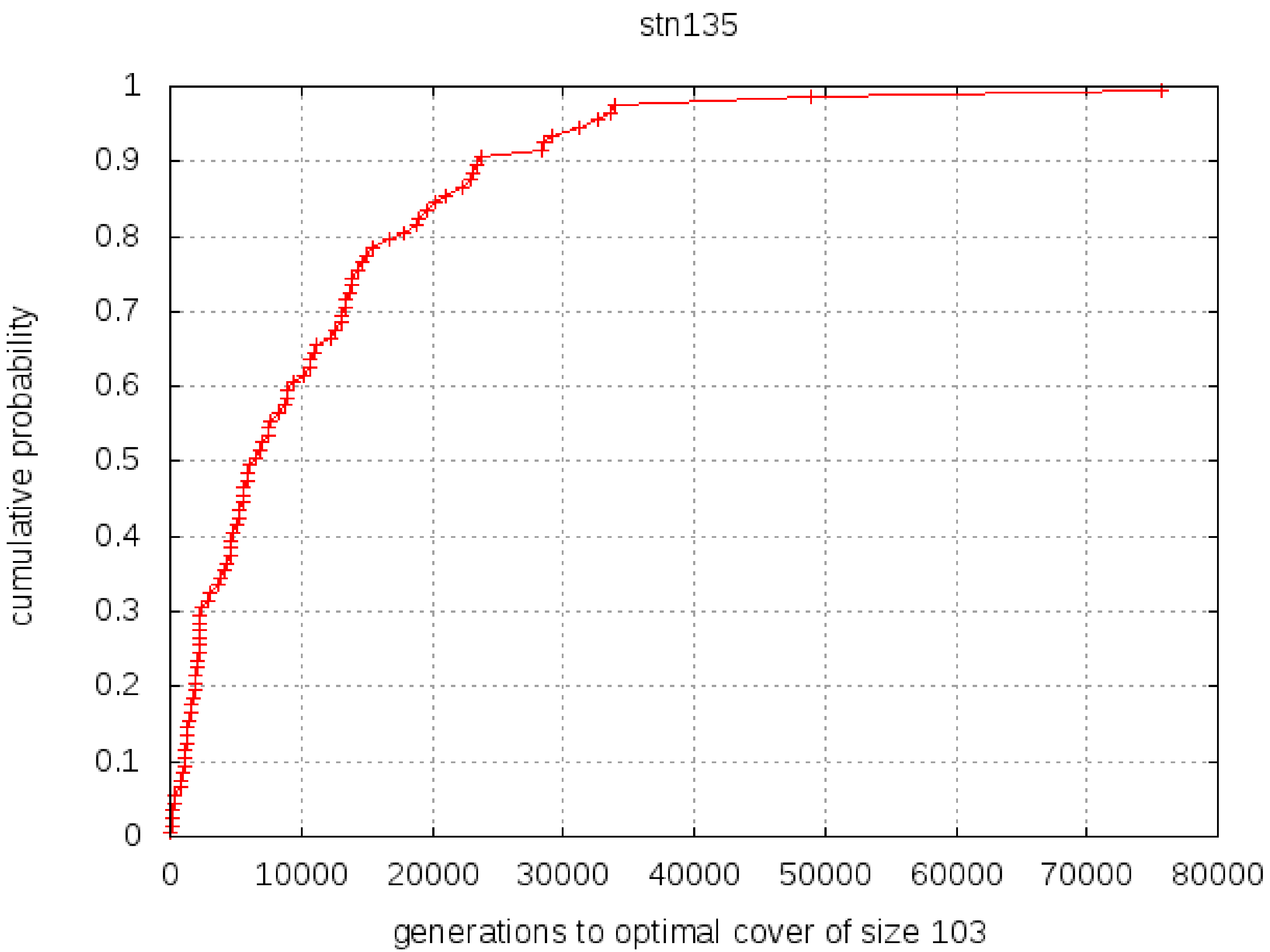
Optimal cover found in initial population in 54/100 runs



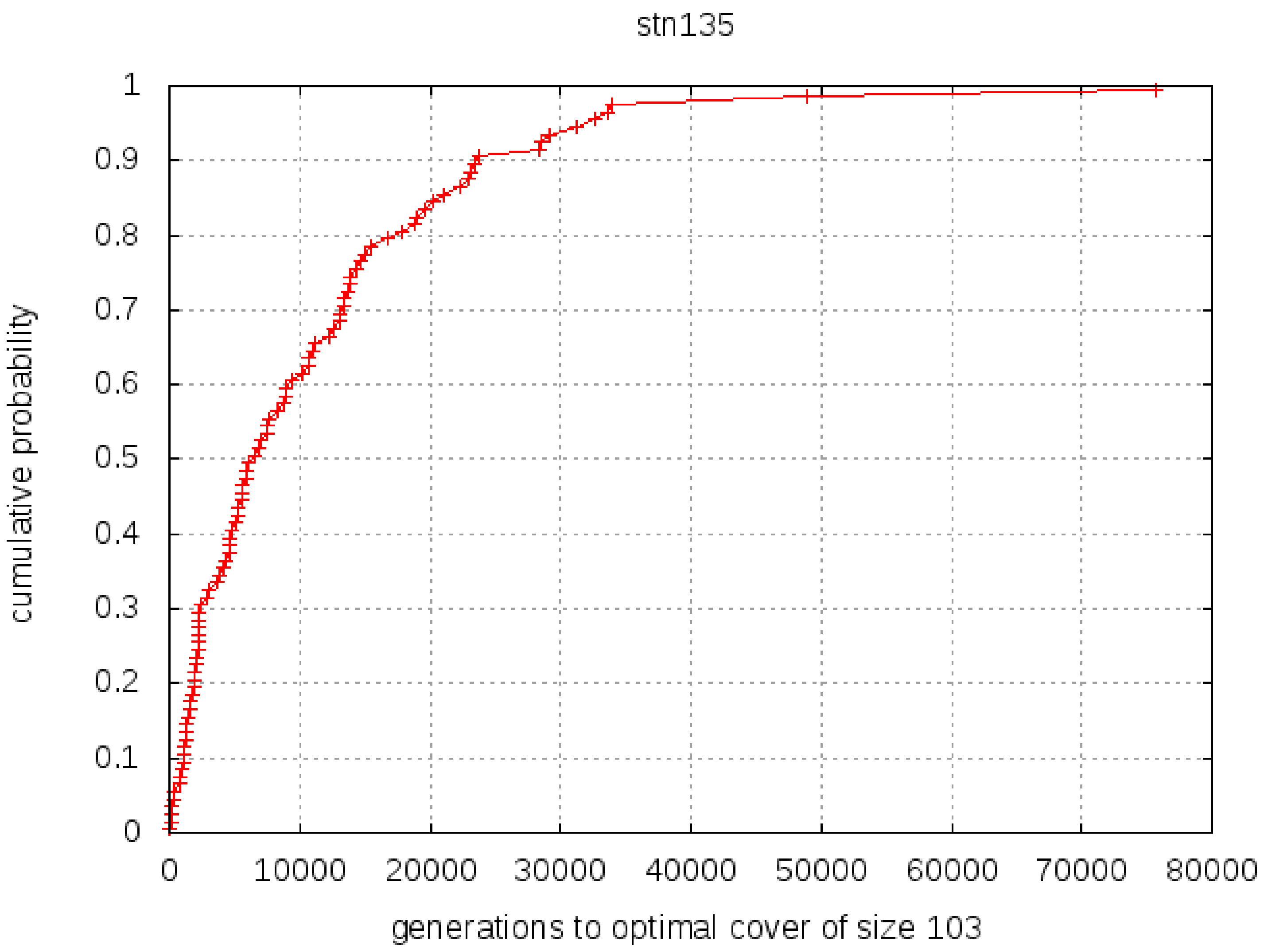
Largest number of iterations in 100 runs was 12



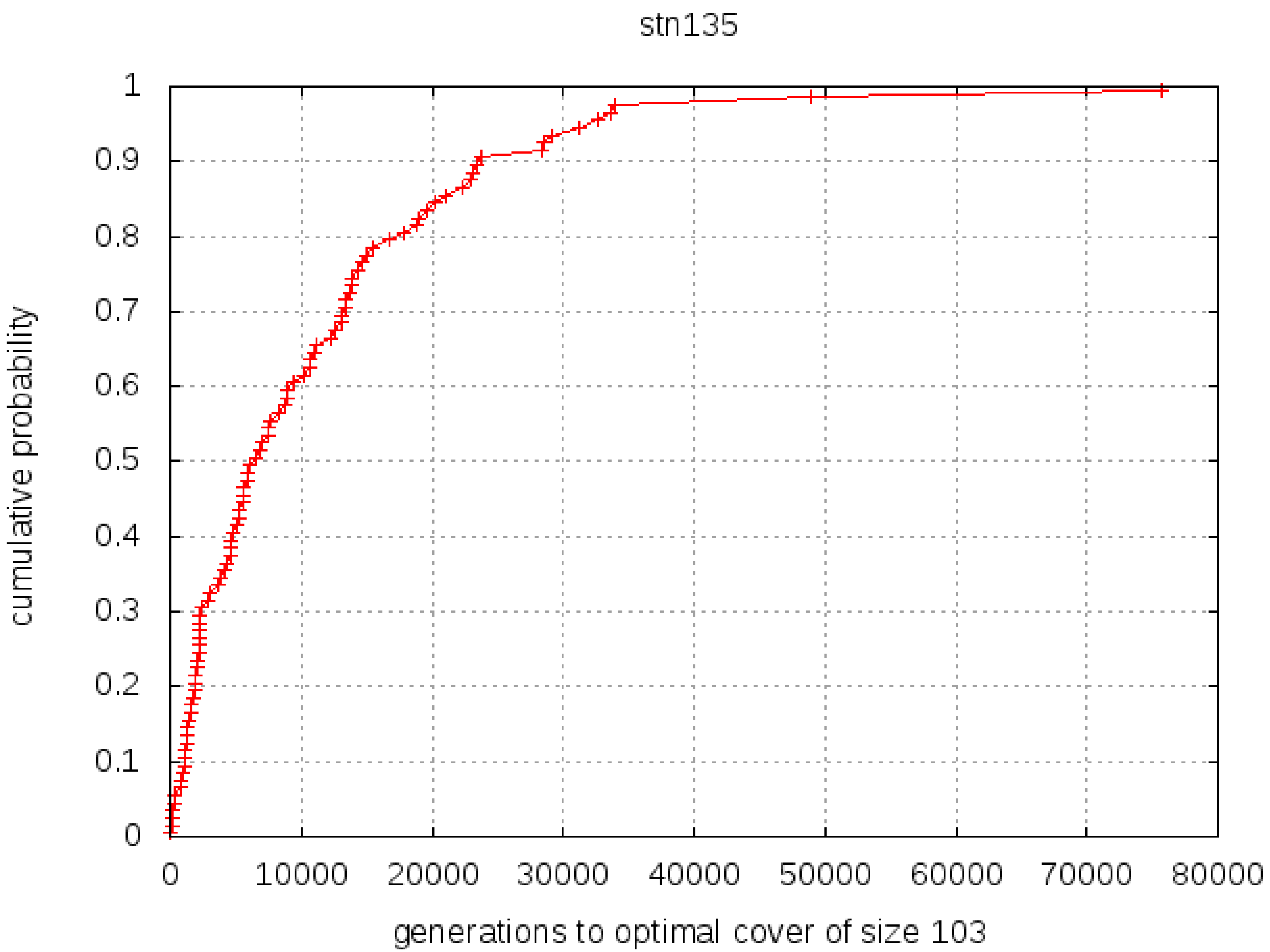
Time per 1000 generations: 4.70s (real), 70.55s (user), 2.73s (sys)



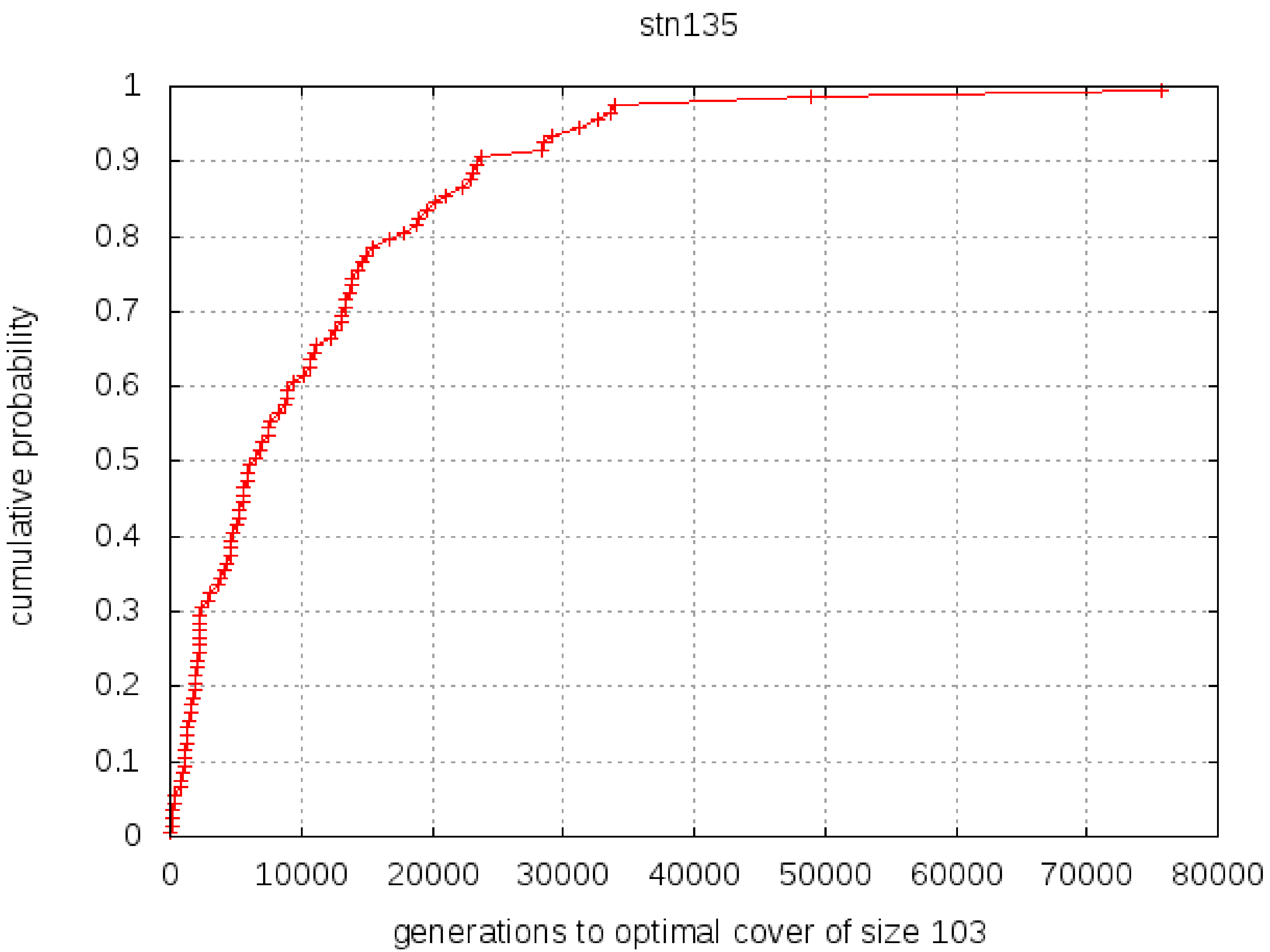
Instance stn135



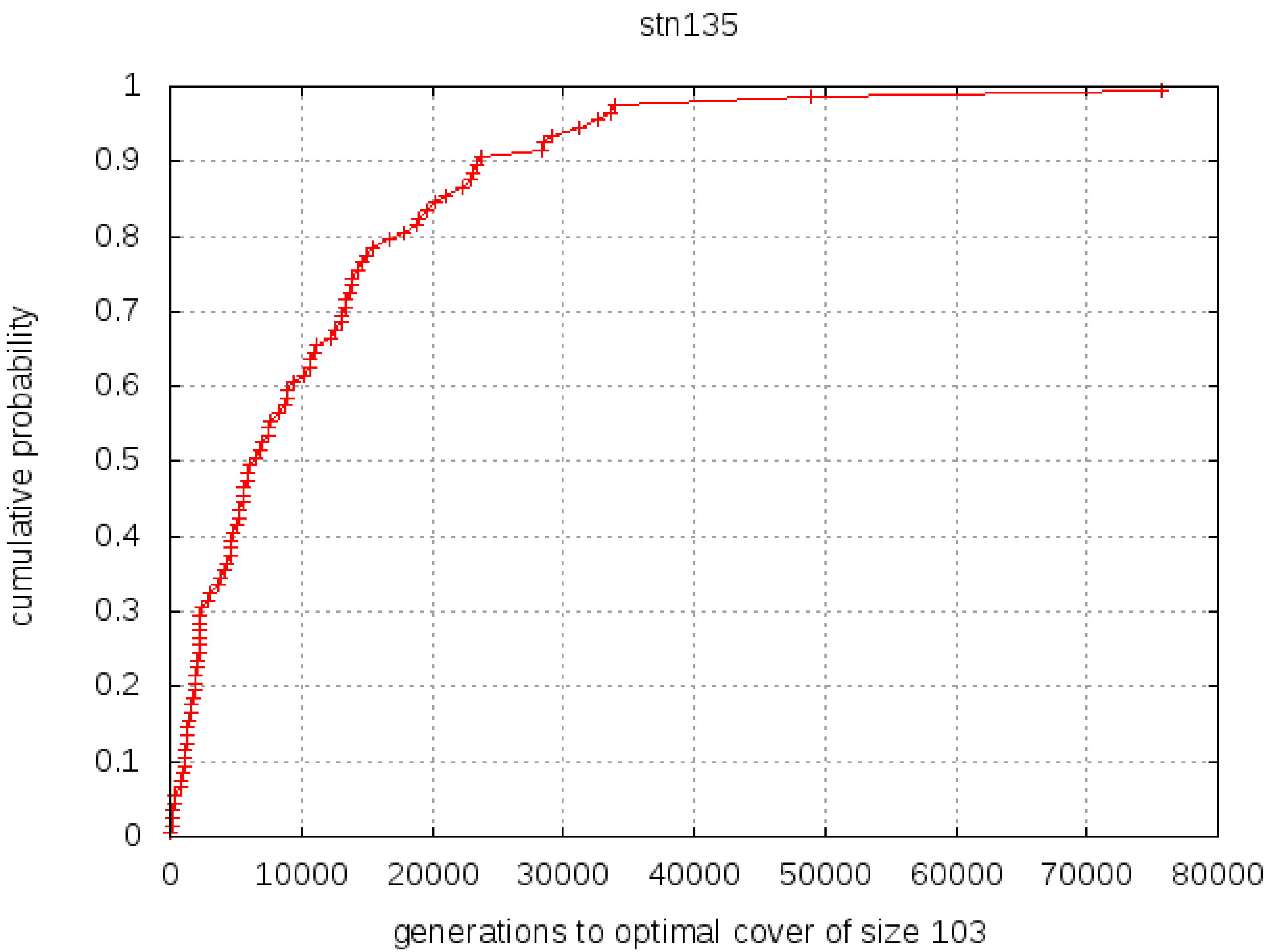
Most difficult instance of those with known optimal cover



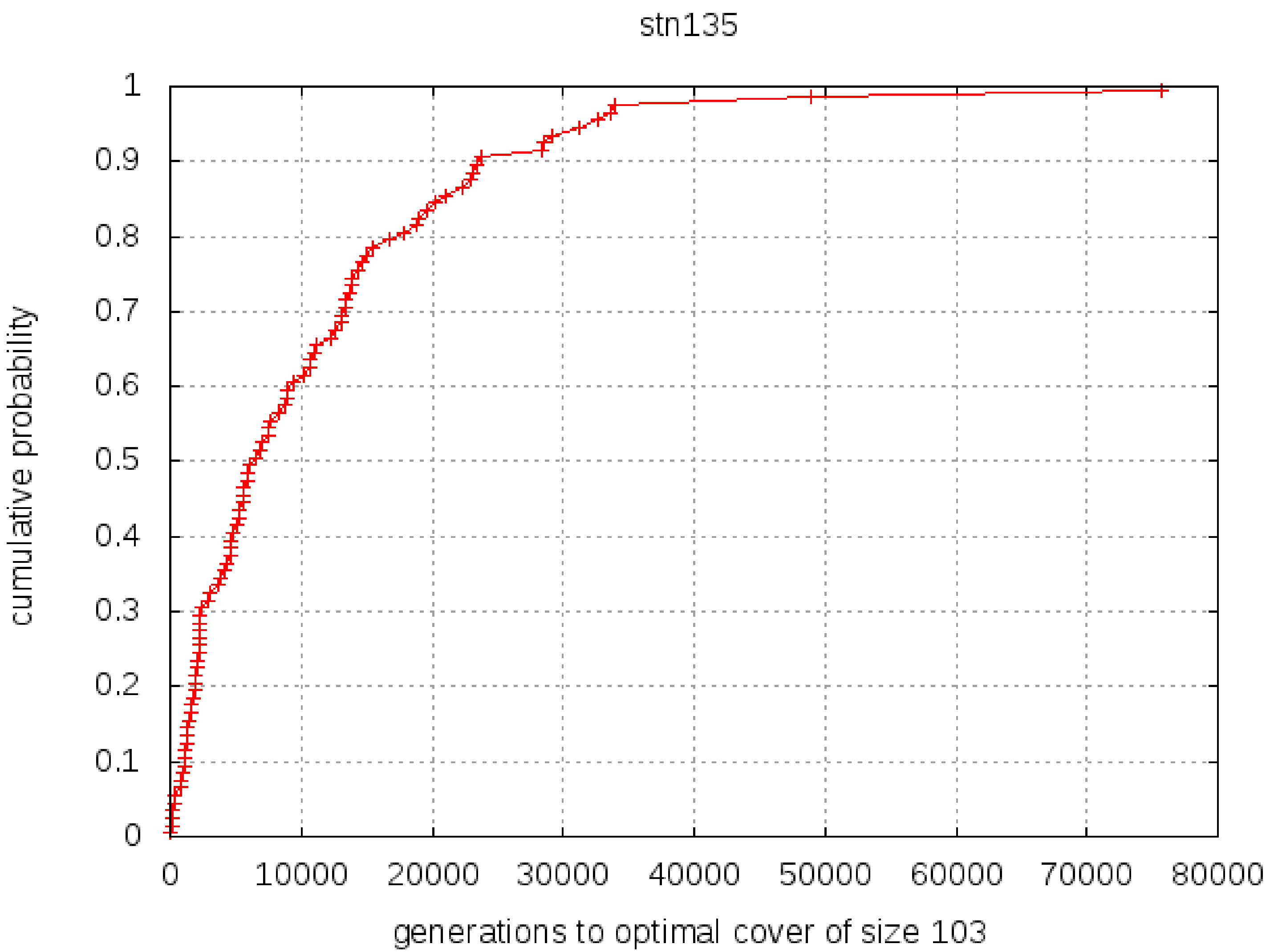
9 of the 100 runs found an optimal cover in less than 1000 iterations



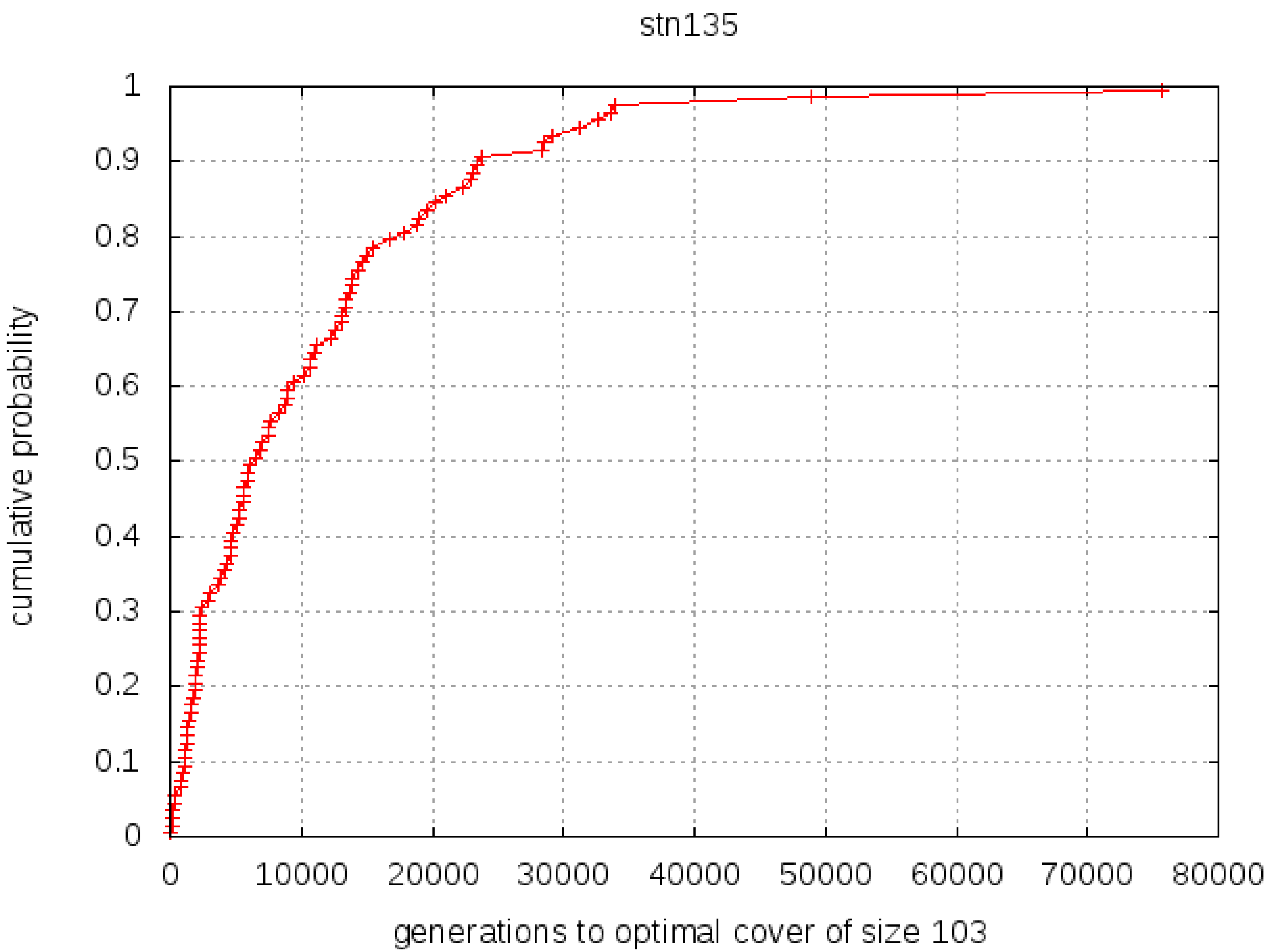
39 of the 100 runs required over 10,000 iterations



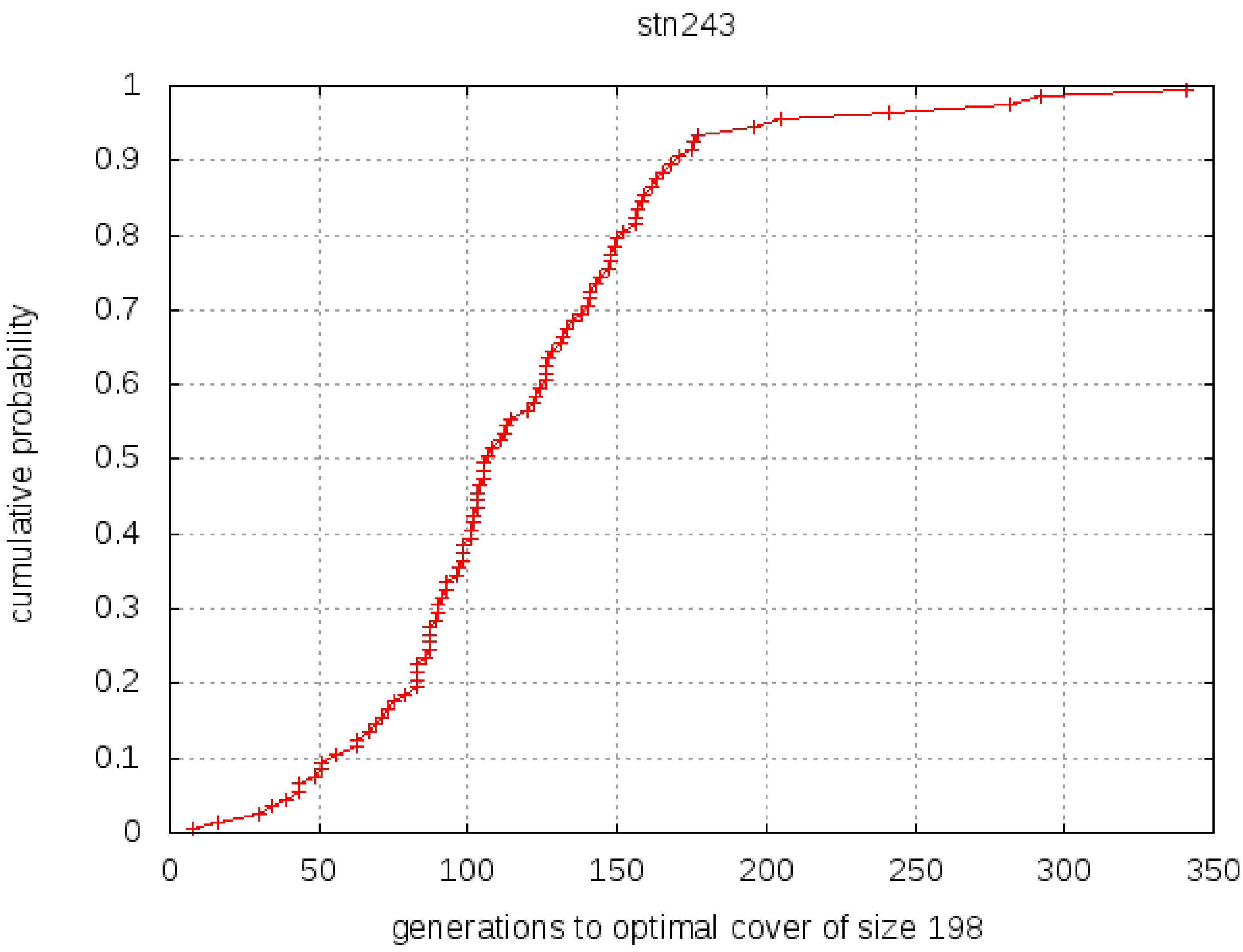
No run required fewer than 23 iterations



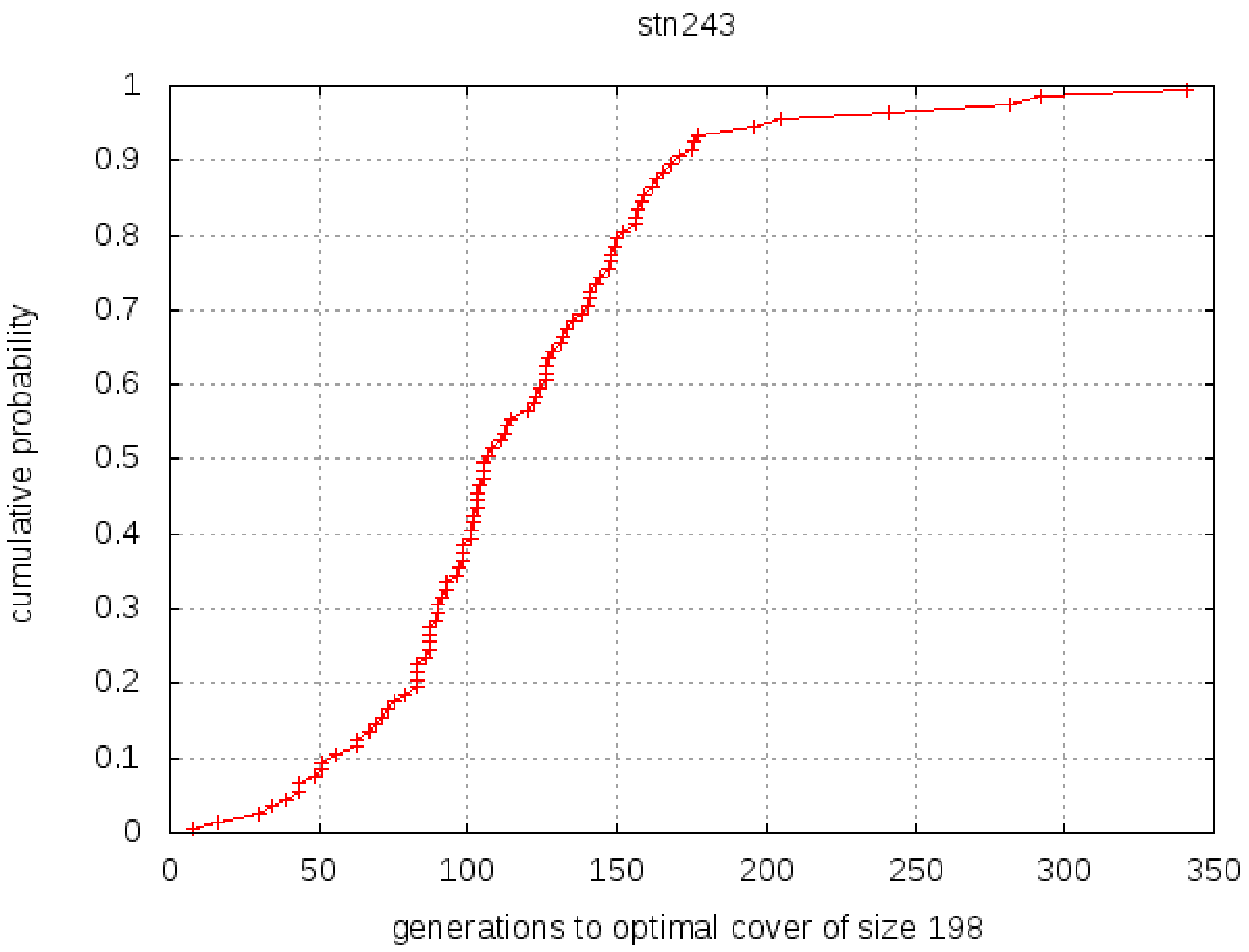
Longest run took 75,741 iterations



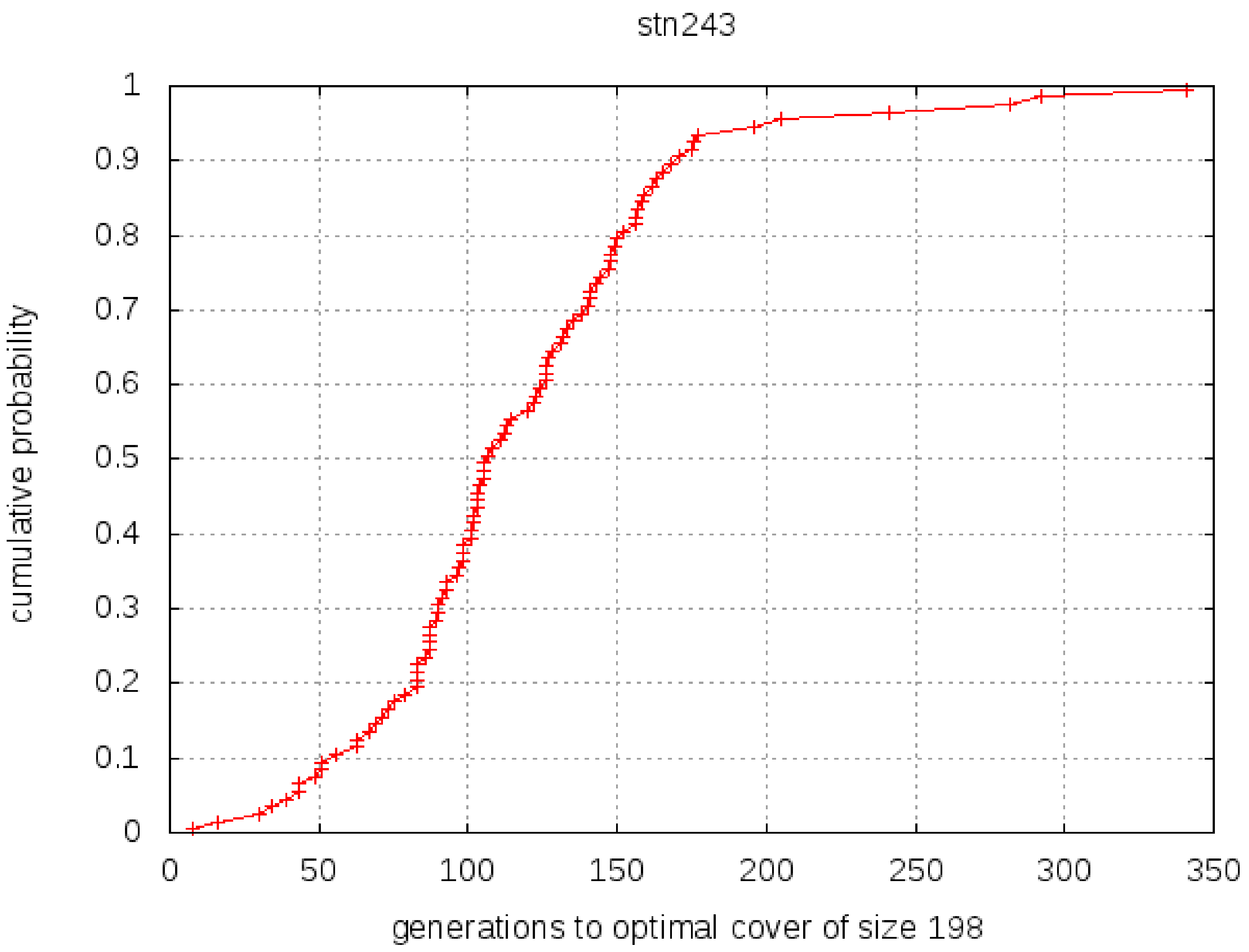
Time per 1000 generations: 19.91s (real), 316.70s (user), 0.85s (sys)



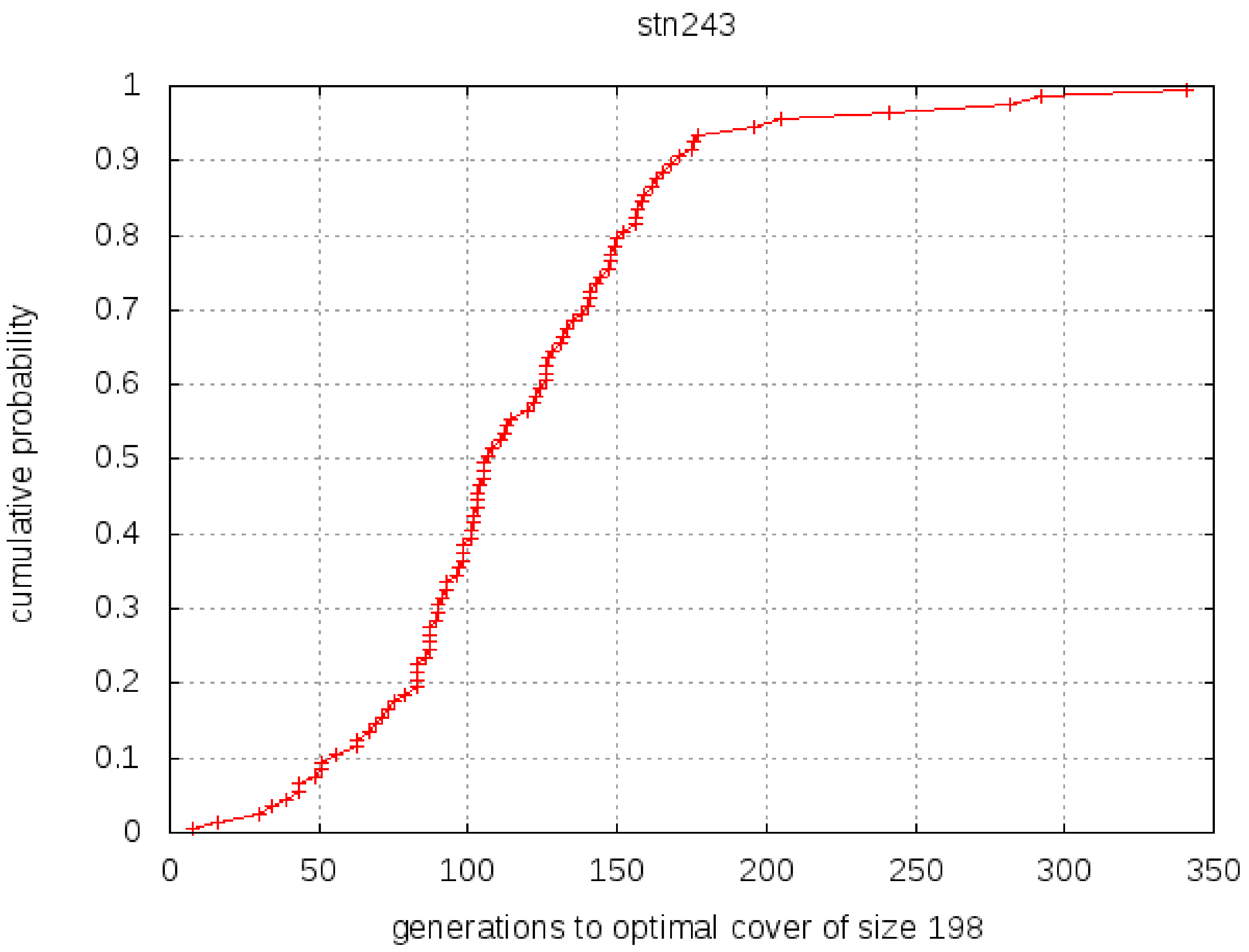
Instance stn243



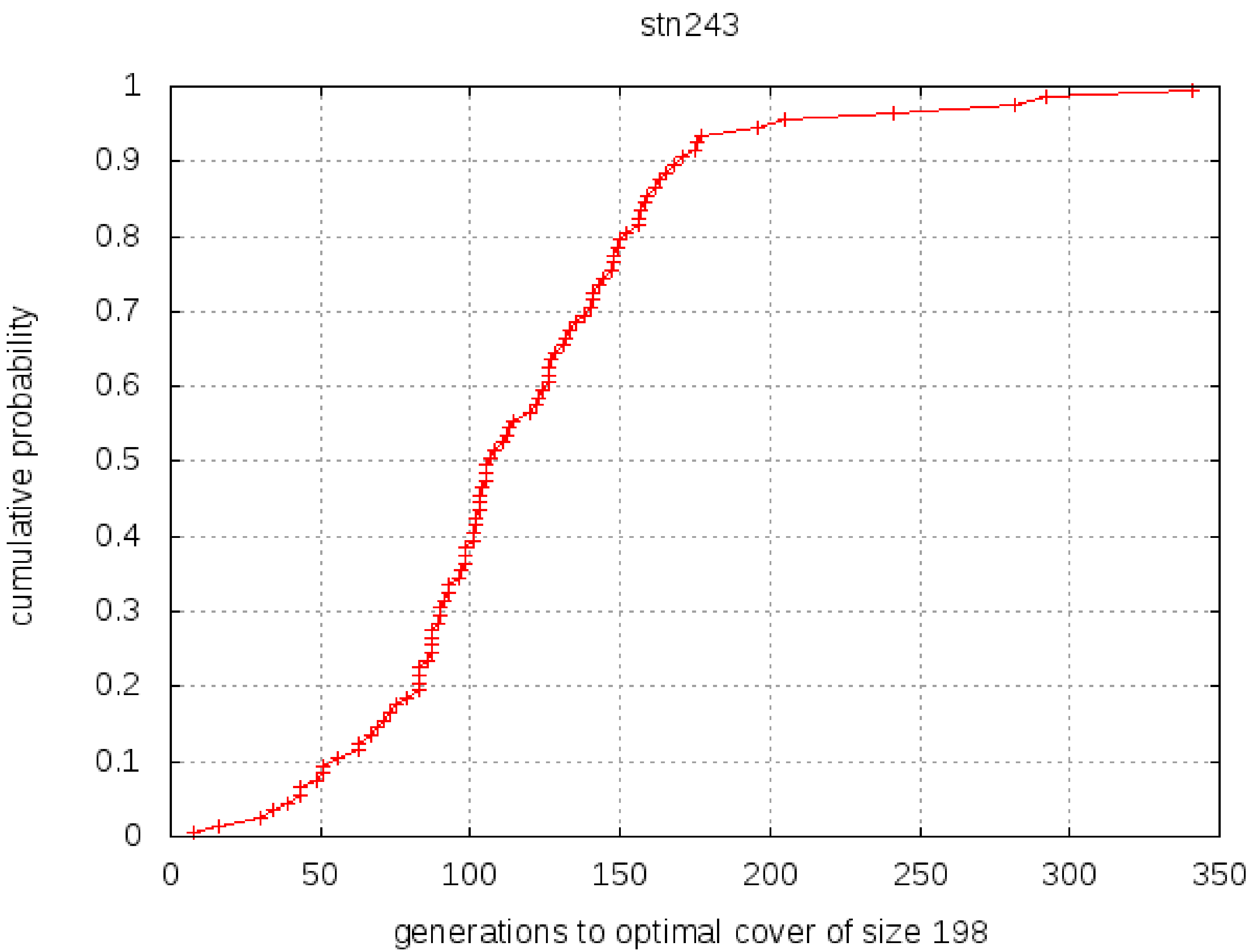
Appears to be much easier than stn135



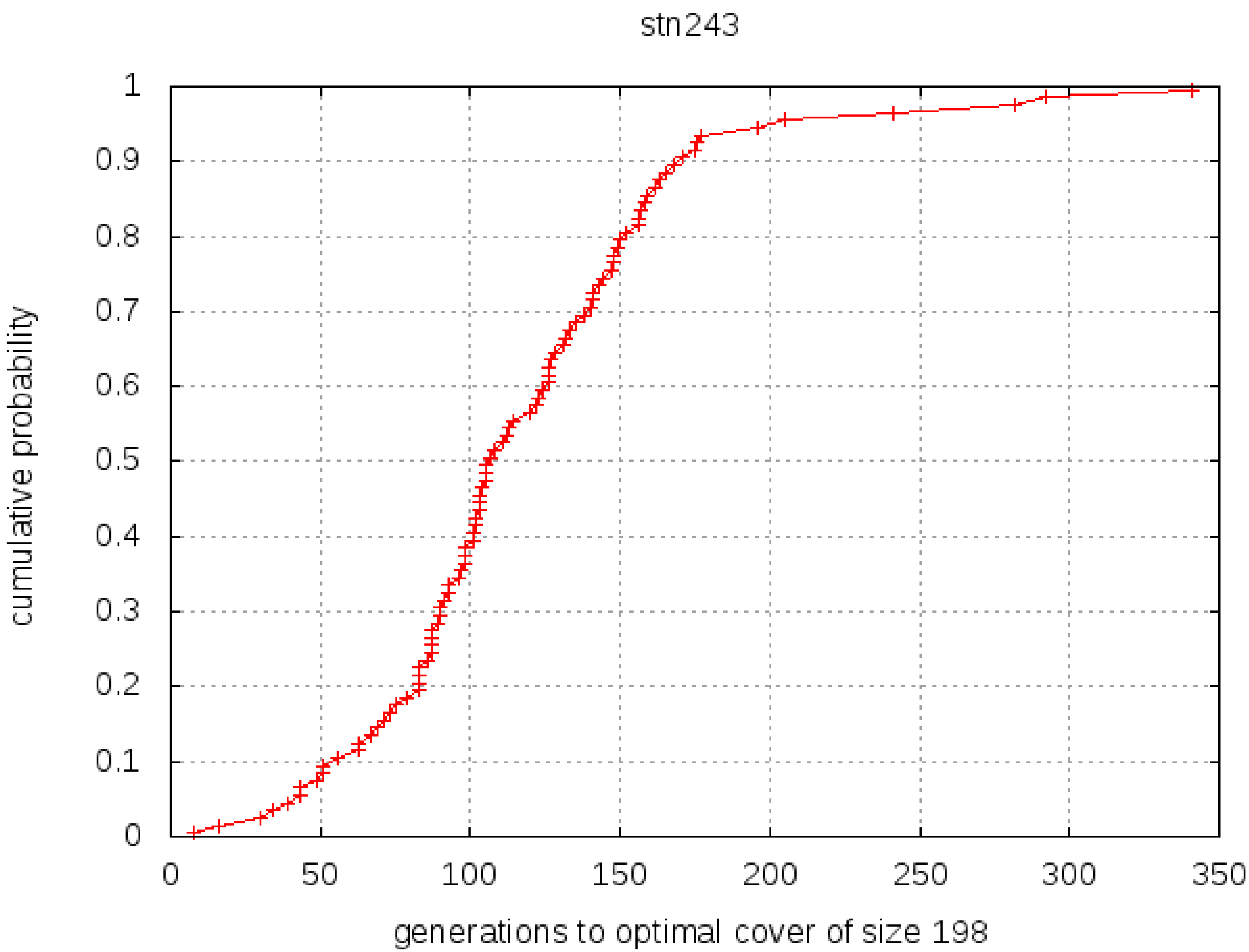
39/100 runs required fewer than 100 generations



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.

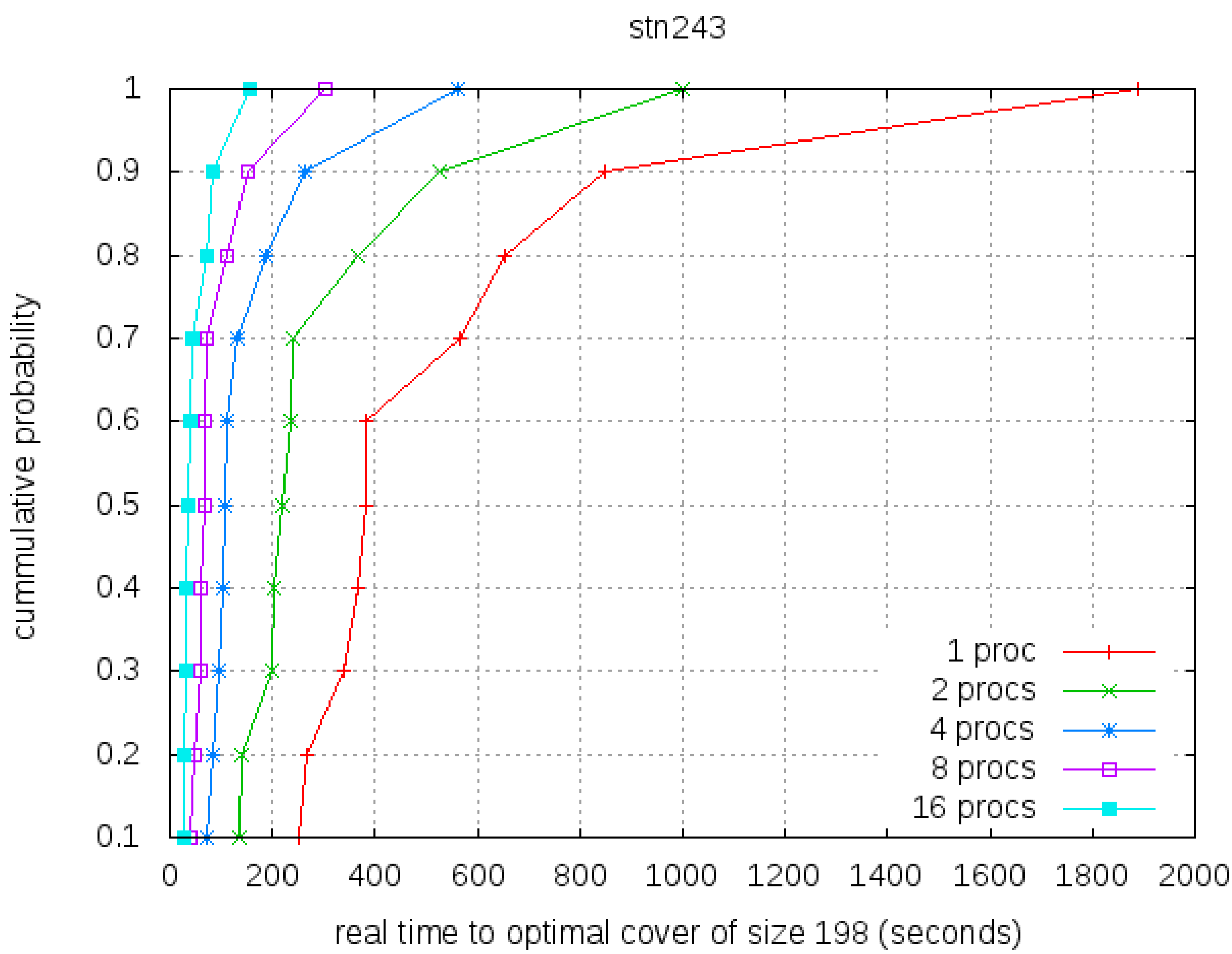
Nevertheless, we observe significant speedup.

Computing covers with a parallel implementation

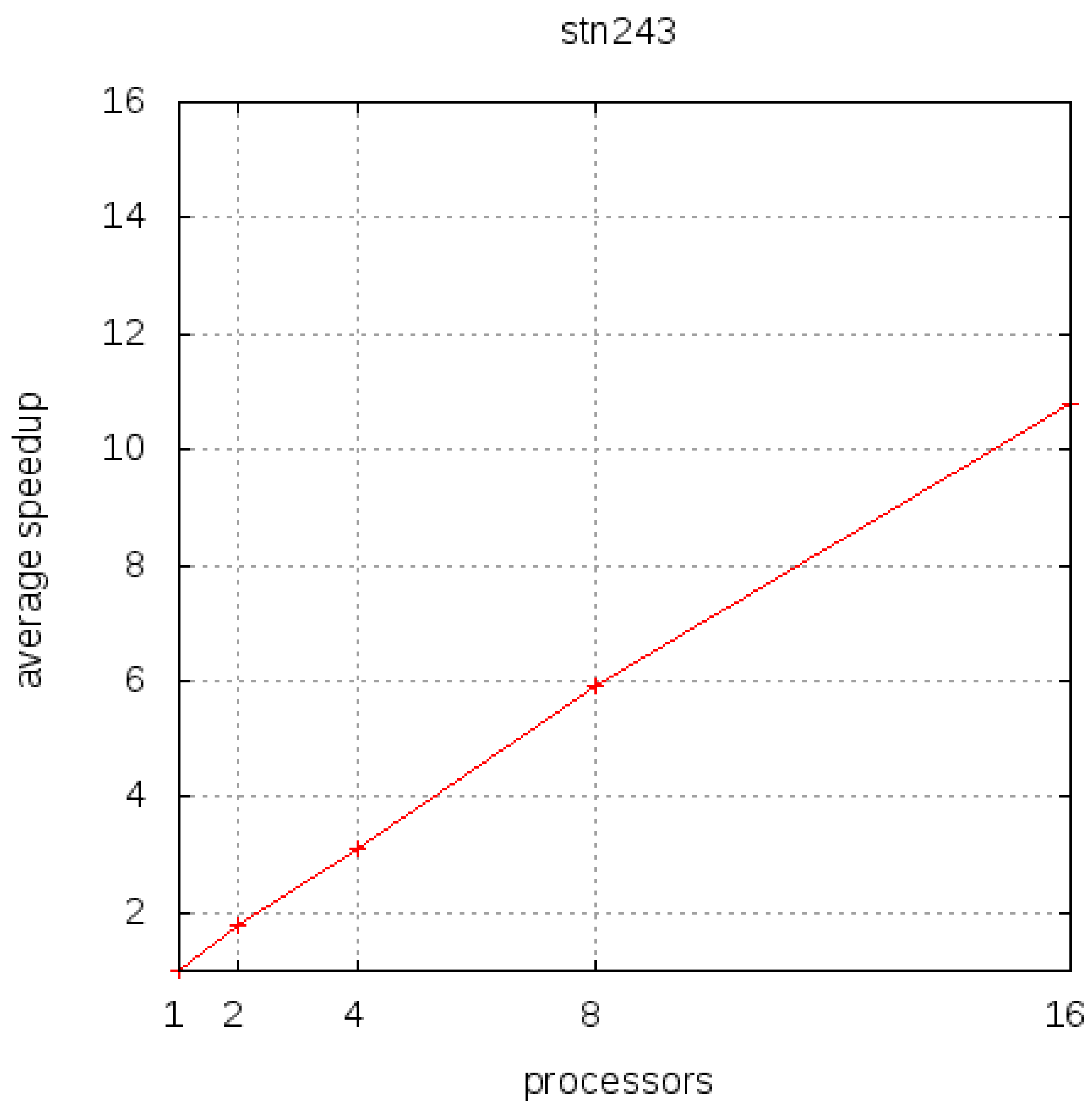
To illustrate the parallel efficiency of the BRKGA we carried out the following experiment on instance stn243 ...

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

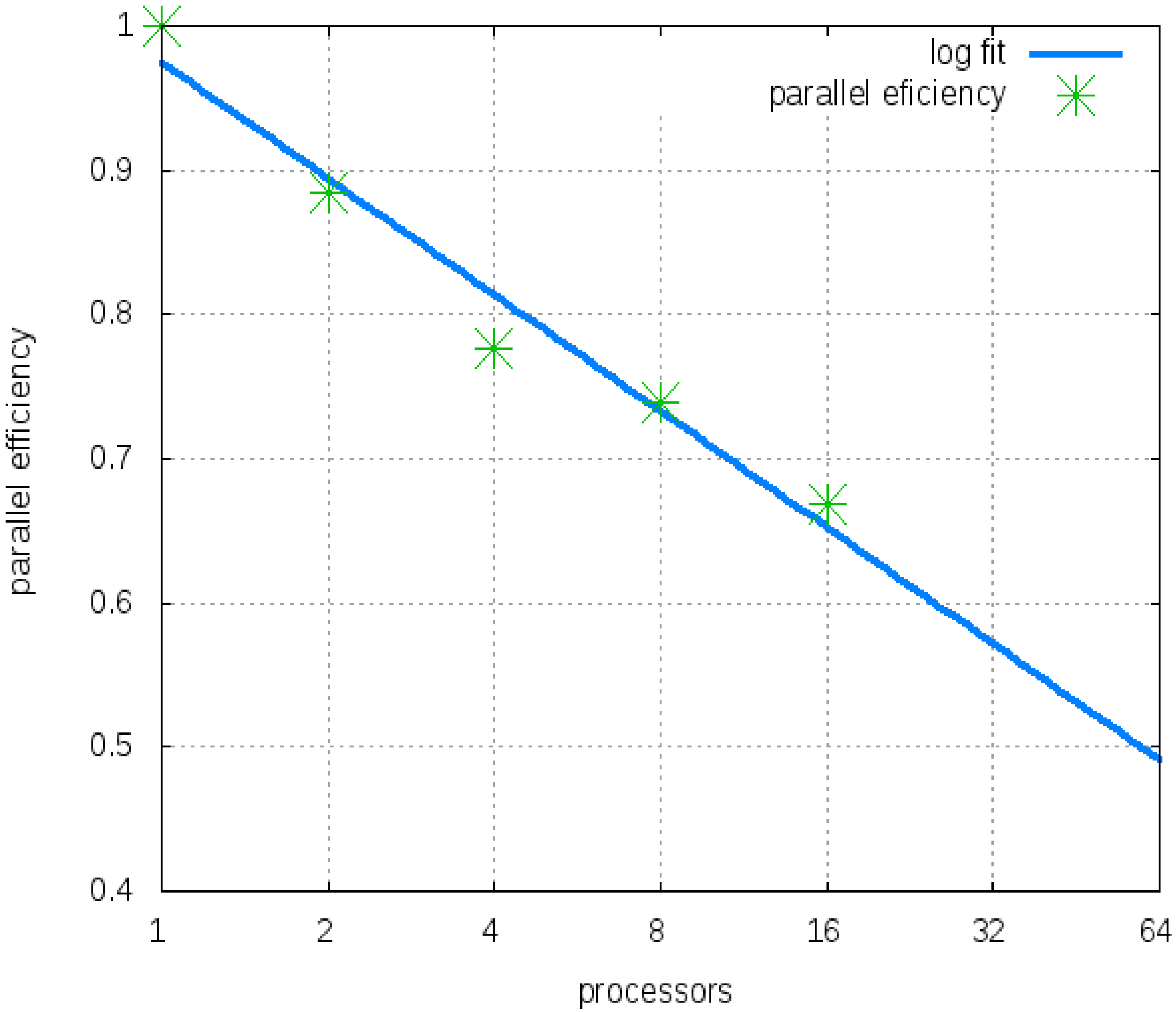
We made 10 independent runs of the BRKGA, stopping when an optimal cover of size 198 was found.



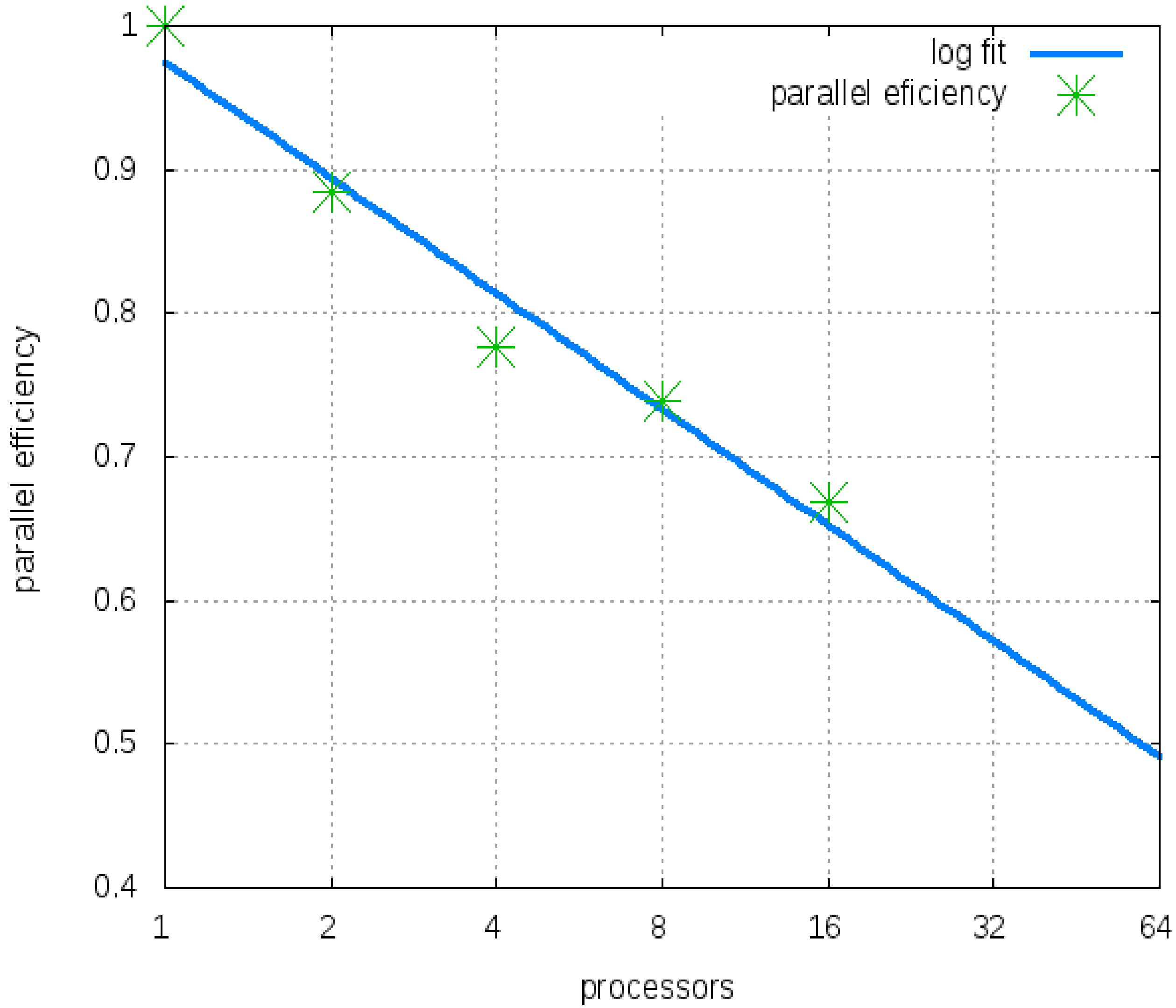
stn243
cummulative probability
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
200
400
600
800
1000
1200
1400
1600
1800
2000
real time to optimal cover of size 198 (seconds)
1 proc
2 procs
4 procs
8 procs
16 procs



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.

Some remarks

Some remarks

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

Some 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 ...

Some 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

Some remarks

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

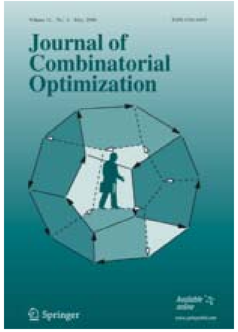
The parallel, multi-population, implementation of the BRKGA not only found optimal covers for all instances with known optimal solution ...

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

Packing weighted rectangles

Reference



J.F. Gonçalves and M.G.C.R., "A parallel multi-population genetic algorithm for a constrained two-dimensional orthogonal packing problem," Journal of Combinatorial Optimization, vol. 22, pp. 180-201, 2011.

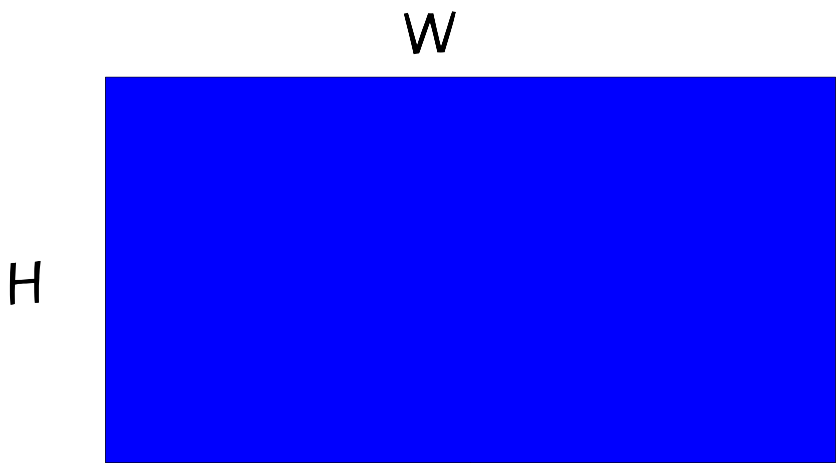
Tech report:
http://www.research.att.com/~mgcr/doc/pack2d.pdf

Constrained orthogonal packing

- Given a large planar stock rectangle (W, H) of width W and height H;

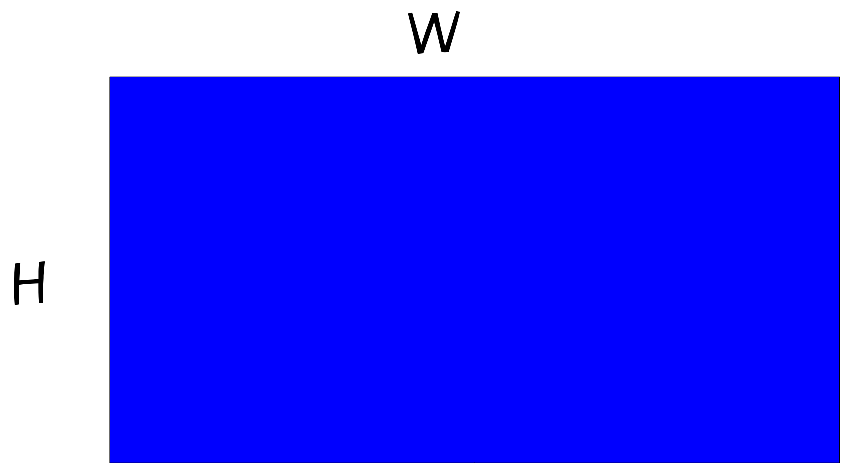
Constrained orthogonal packing

- Given a large planar stock rectangle (W, H) of width W and height H;



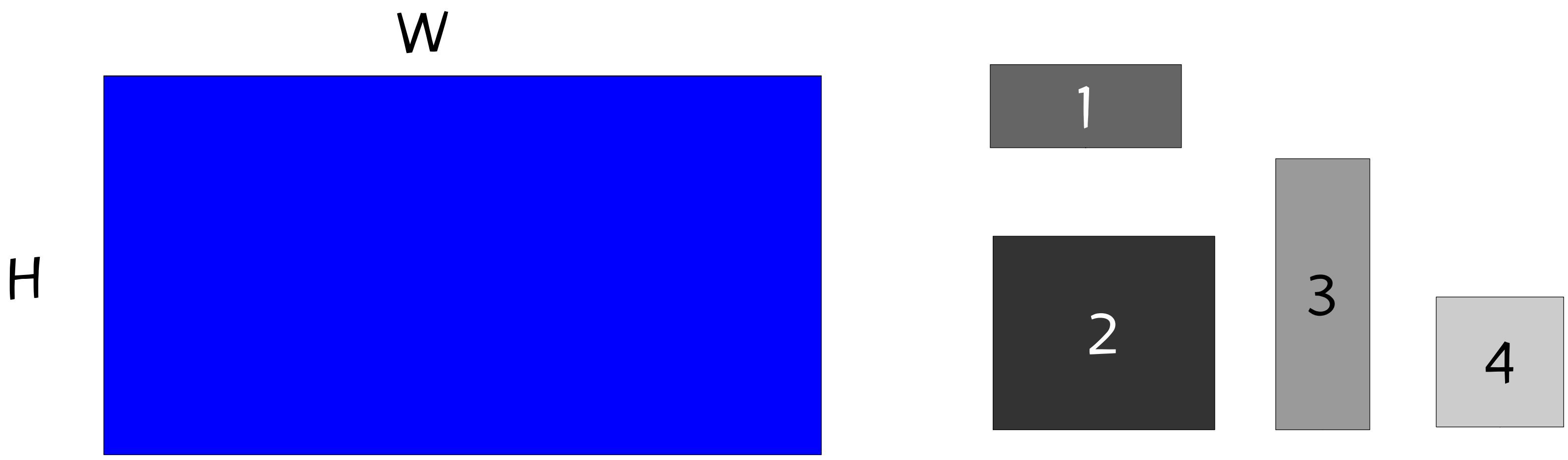
Constrained orthogonal packing

- Given a large planar stock rectangle (W, H) of width W and height H;
- Given N smaller rectangle types (w[i], h[i]), i = 1,...,N, each of width w[i], height h[i], and value v[i];



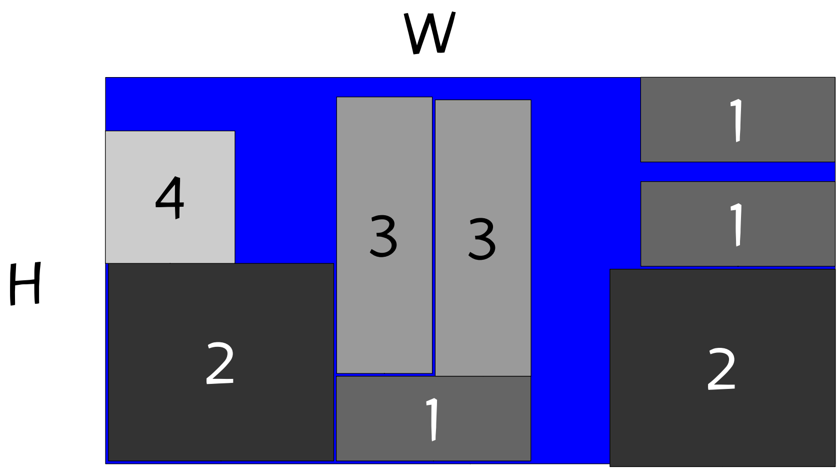
Constrained orthogonal packing

- Given a large planar stock rectangle (W, H) of width W and height H;
- Given N smaller rectangle types (w[i], h[i]), i = 1,…,N, each of width w[i], height h[i], and value v[i];



Constrained orthogonal packing

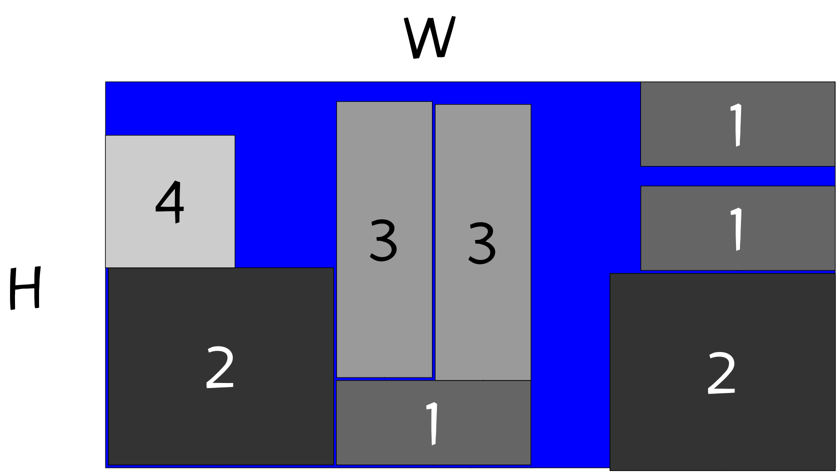
- r[i] rectangles of type i = 1, ..., N are to be packed in the large rectangle without overlap and such that their edges are parallel to the edges of the large rectangle;



Constrained orthogonal packing

- r[i] rectangles of type i = 1, ..., N are to be packed in the large rectangle without overlap and such that their edges are parallel to the edges of the large rectangle;
- For i = 1, ..., N, we require that:

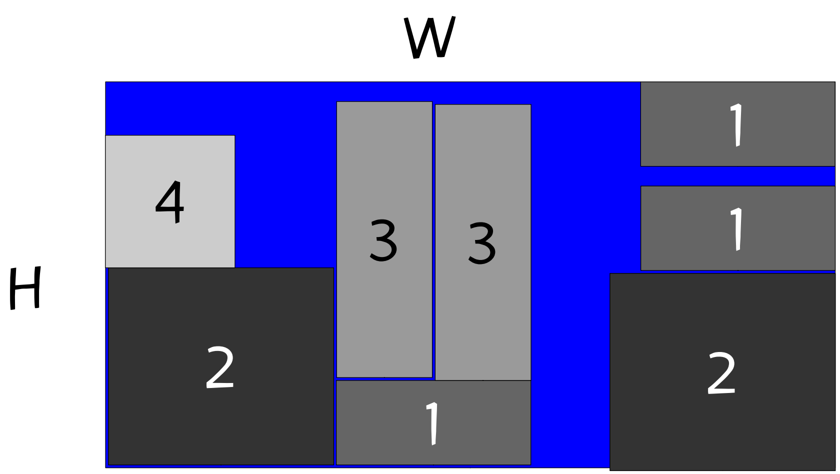
$$0 \leq P[i] \leq r[i] \leq Q[i]$$



Constrained orthogonal packing

- r[i] rectangles of type i = 1, ..., N are to be packed in the large rectangle without overlap and such that their edges are parallel to the edges of the large rectangle;
- For i = 1, ..., N, we require that:

$$0 \leq P[i] \leq r[i] \leq Q[i]$$

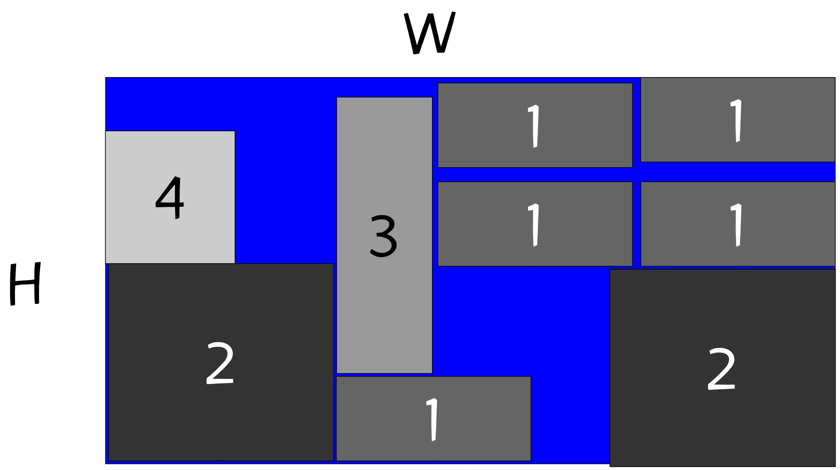


Suppose $5 \leq r[1] \leq 12$

Constrained orthogonal packing

- r[i] rectangles of type i = 1, ..., N are to be packed in the large rectangle without overlap and such that their edges are parallel to the edges of the large rectangle;
- For i = 1, ..., N, we require that:

$$0 \leq P[i] \leq r[i] \leq Q[i]$$

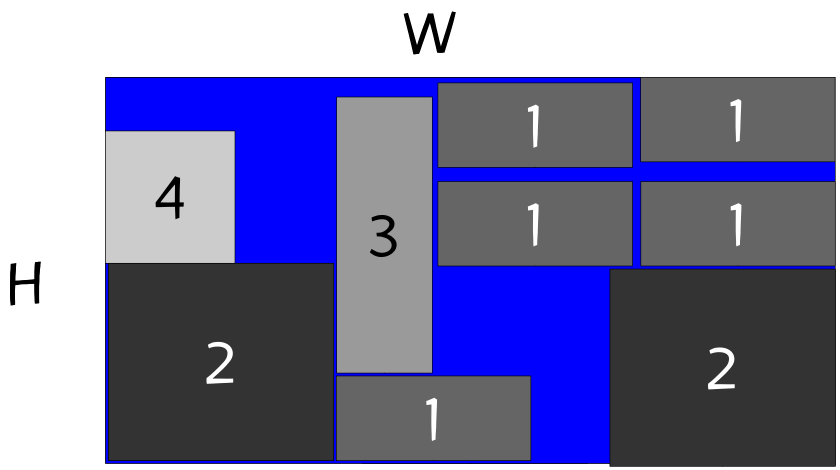


Suppose $5 \leq r[1] \leq 12$

Objective

Among the many feasible packings, we want to find one that maximizes total value of packed rectangles:

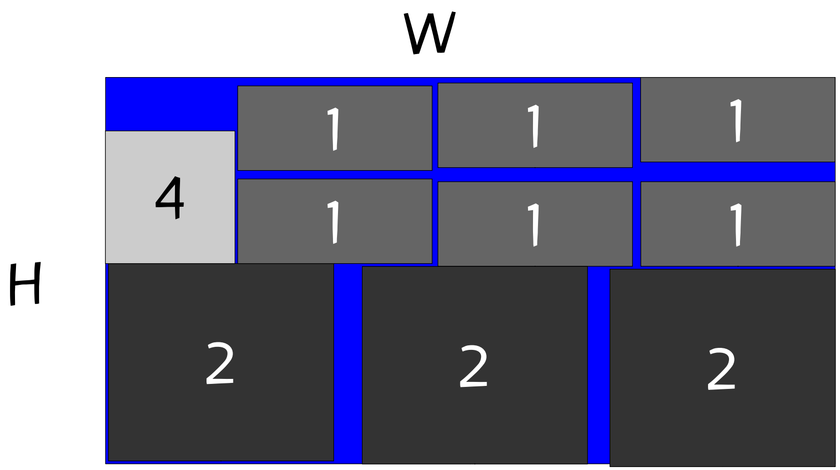
$$v[1]\ r[1] + v[2]\ r[2] + \cdots + v[N]\ r[N]$$



Objective

Among the many feasible packings, we want to find one that maximizes total value of packed rectangles:

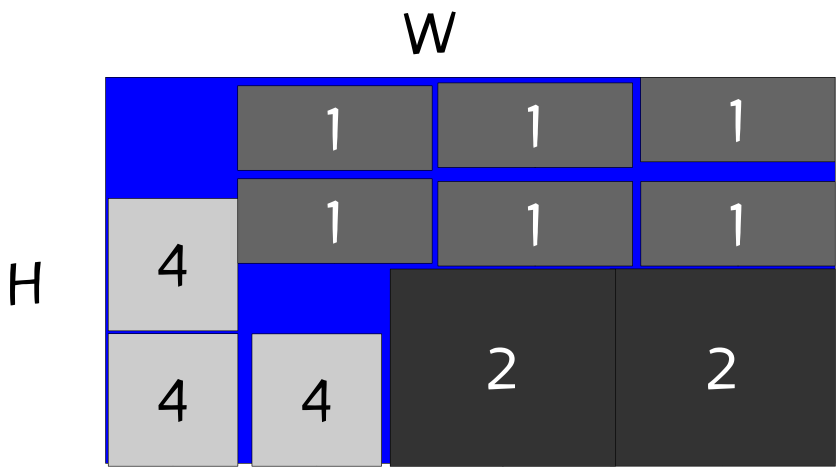
$$v[1]\ r[1] + v[2]\ r[2] + \cdots + v[N]\ r[N]$$



Objective

Among the many feasible packings, we want to find one that maximizes total value of packed rectangles:

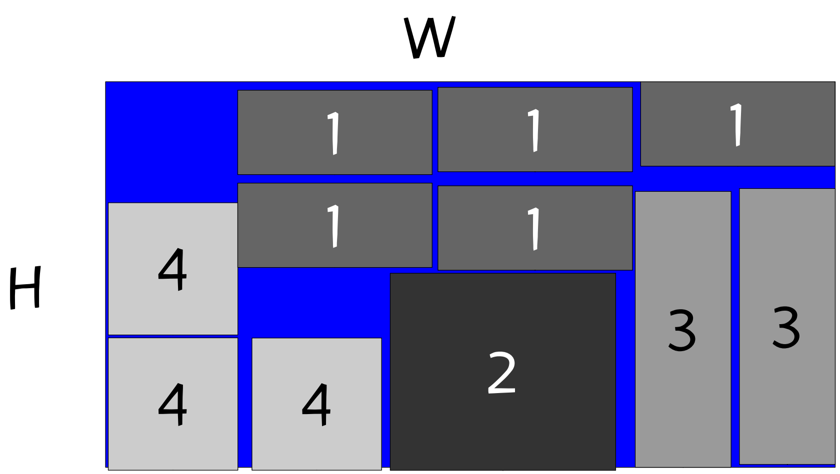
$$v[1]\ r[1] + v[2]\ r[2] + \cdots + v[N]\ r[N]$$



Objective

Among the many feasible packings, we want to find one that maximizes total value of packed rectangles:

v[1] r[1] + v[2] r[2] + · · · +v[N] r[N]

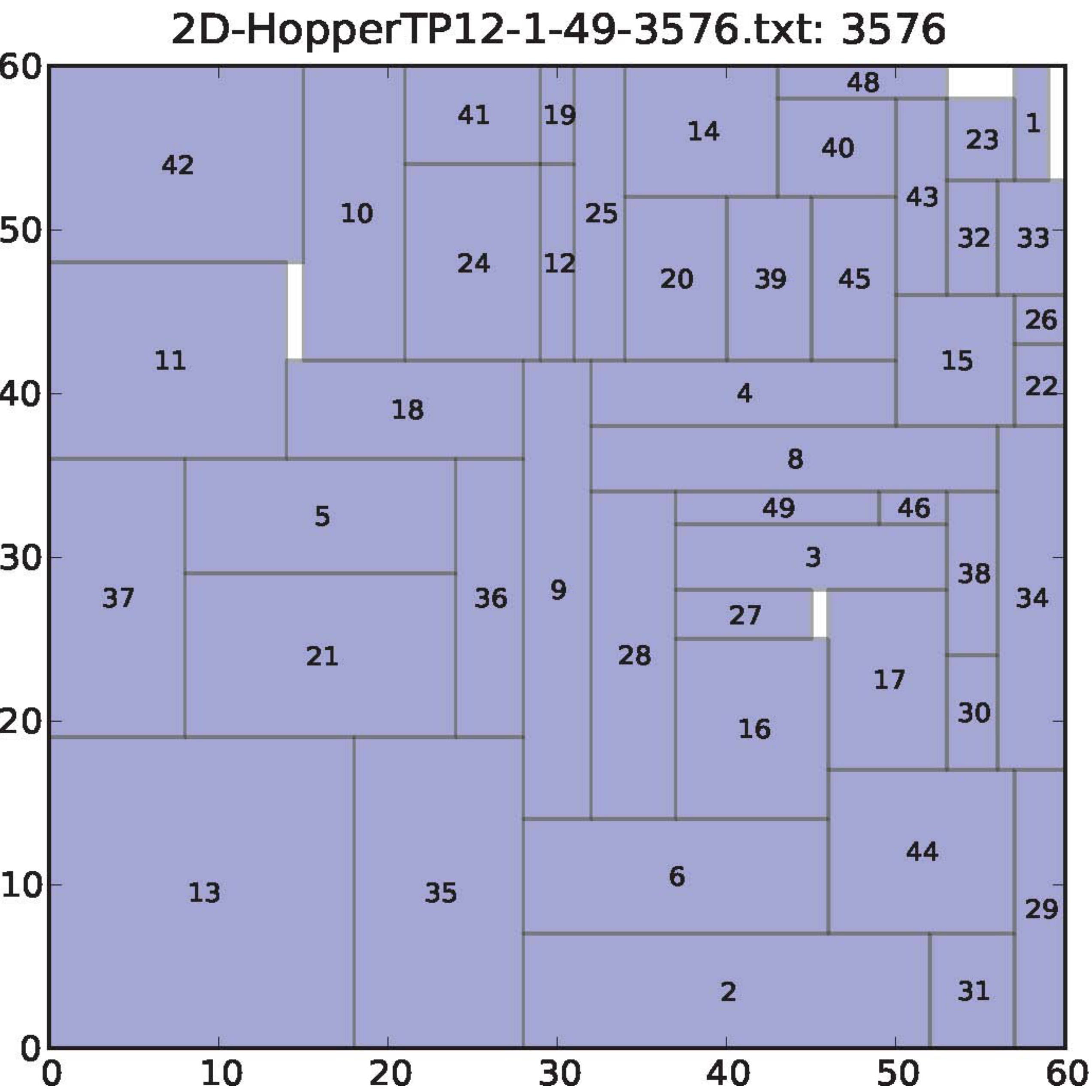


Applications

Problem arises in several production processes, e.g.

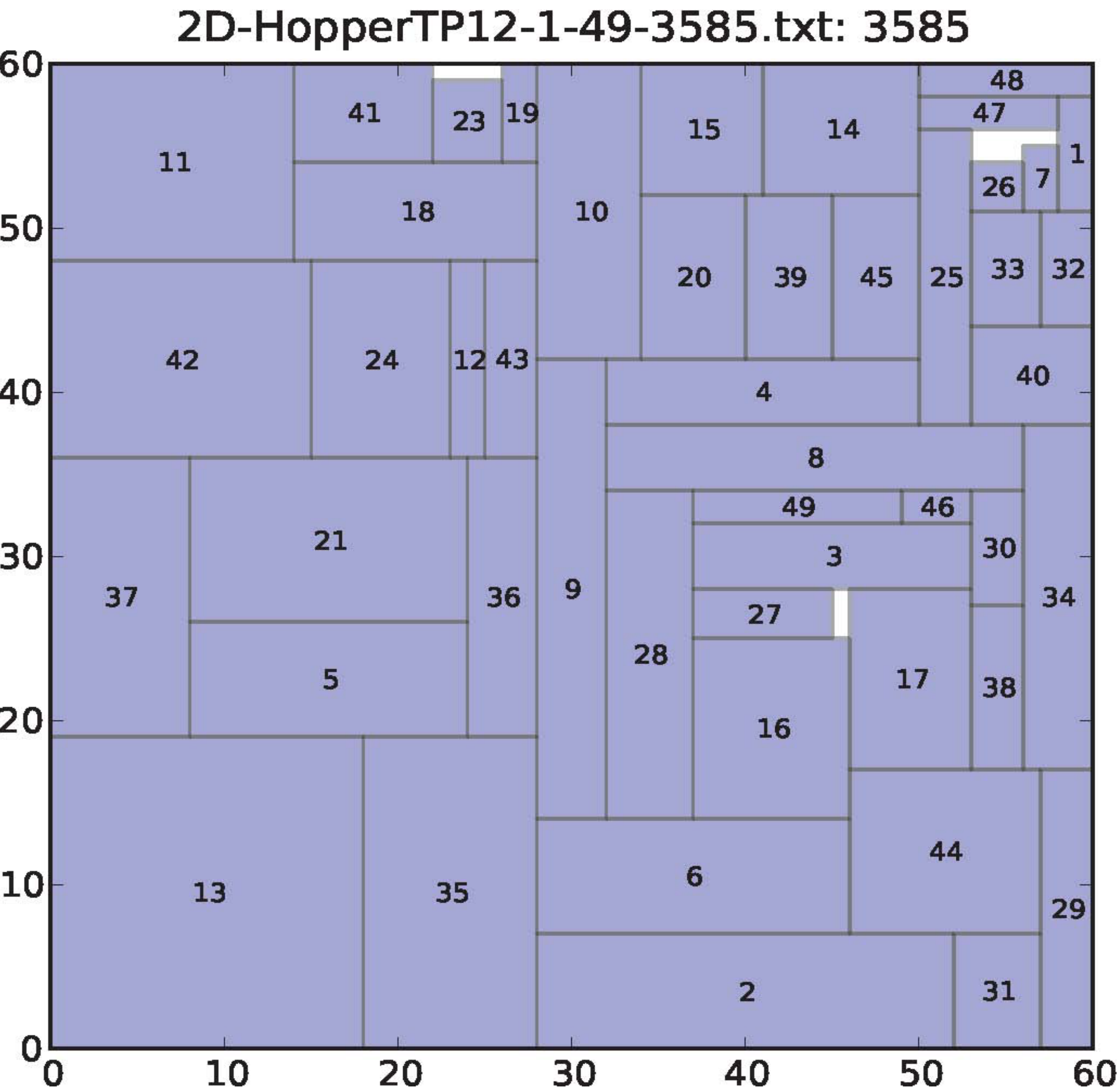
- Textile
- Glass
- Wood
- Paper

where rectangular figures are cut from large rectangular sheets of materials.



Hopper & Turton, 2001
Instance 4-1 60 x 60
Value: 3576

Previous best: 3580 by a Tabu Search heuristic (Alvarez-Valdes et al., 2007)

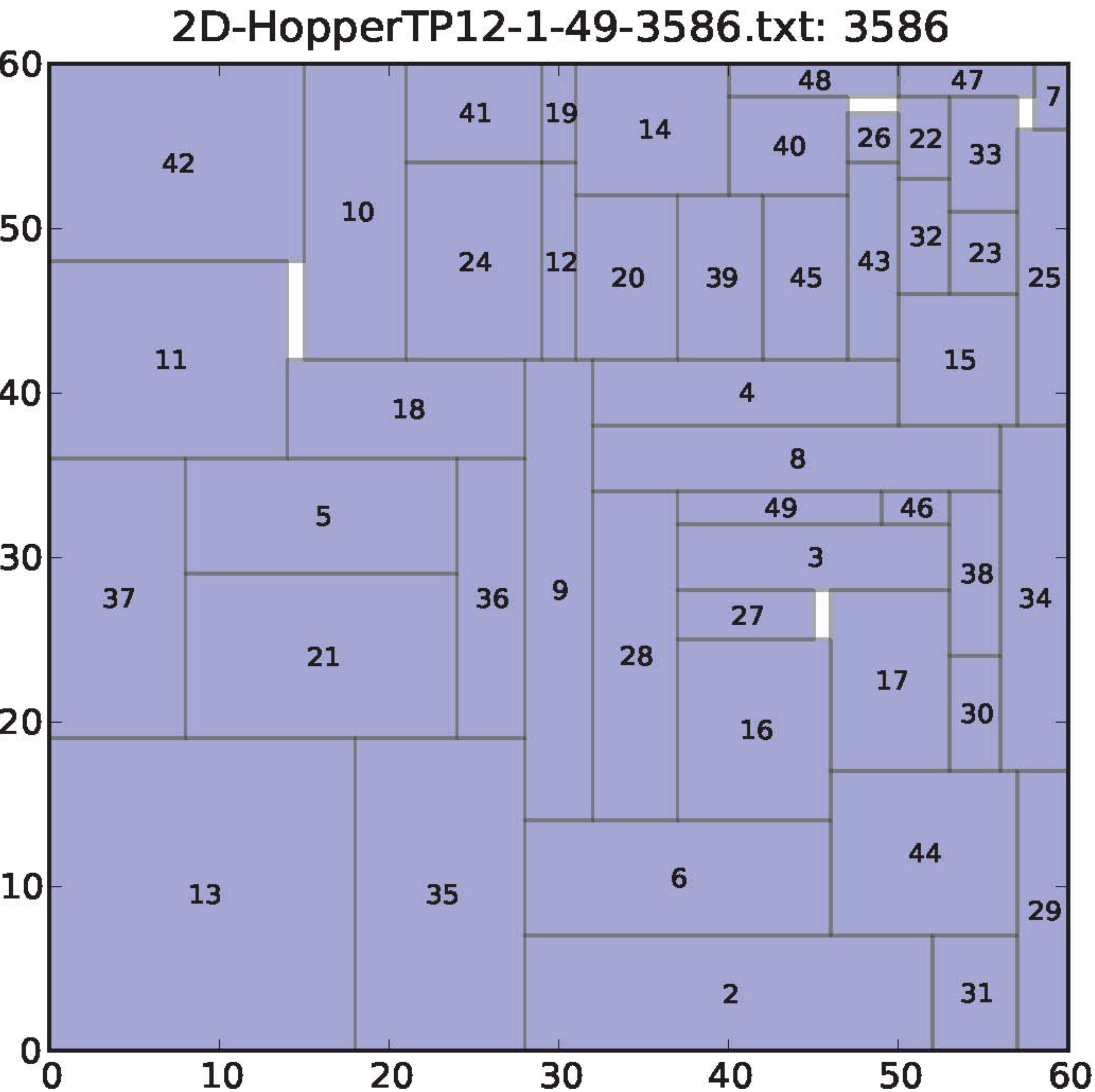


Hopper & Turton, 2001

Instance 4-2 60 x 60

Value: 3585

Previous best: 3580 by a Tabu Search heuristic (Alvarez-Valdes et al., 2007)



Hopper & Turton, 2001

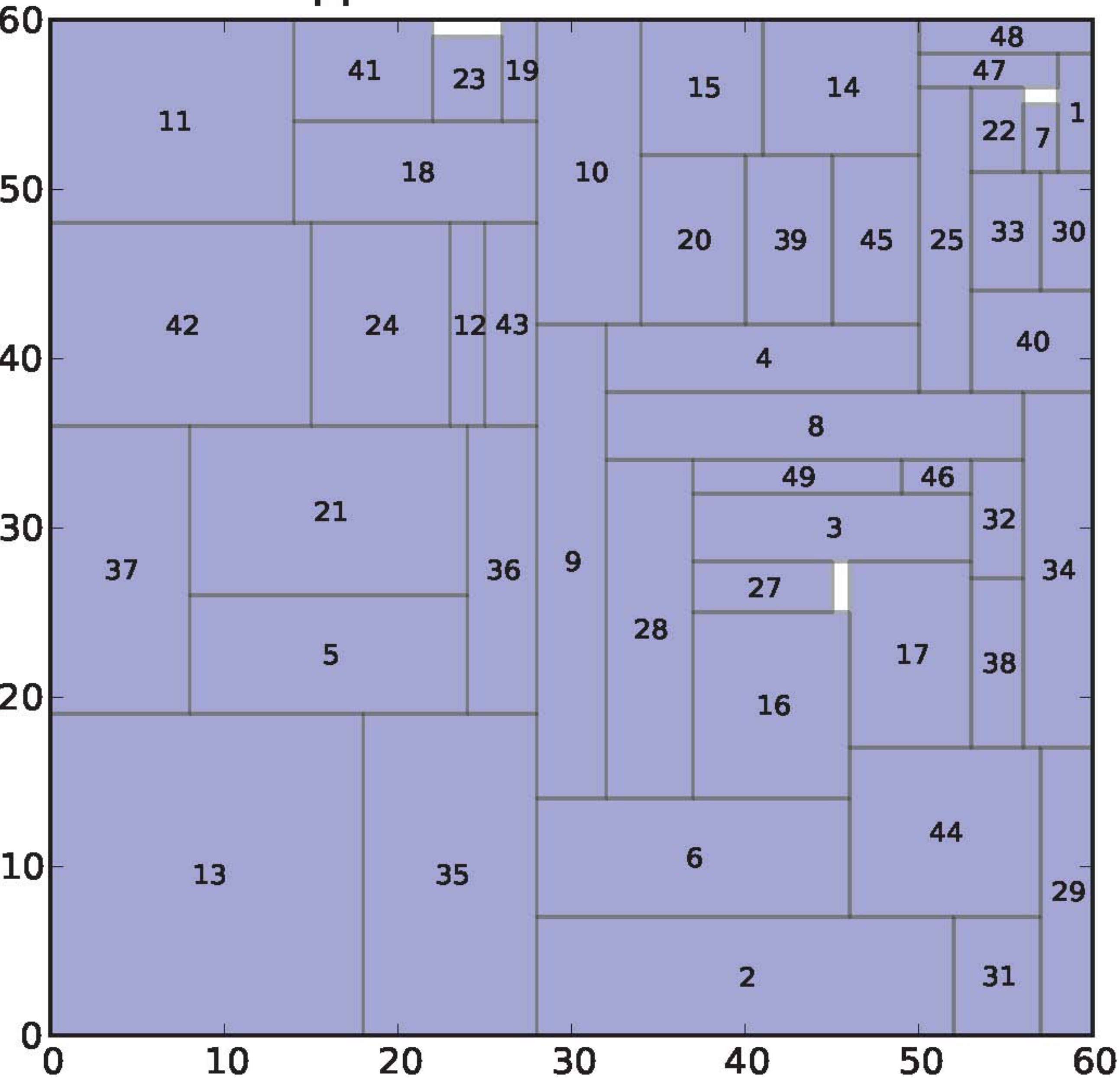
Instance 4-2 60 x 60

Value: 3586

Previous best: 3580 by a Tabu Search heuristic

(Alvarez-Valdes et al., 2007)

2D-HopperTP12-1-49-3591.txt: 3591

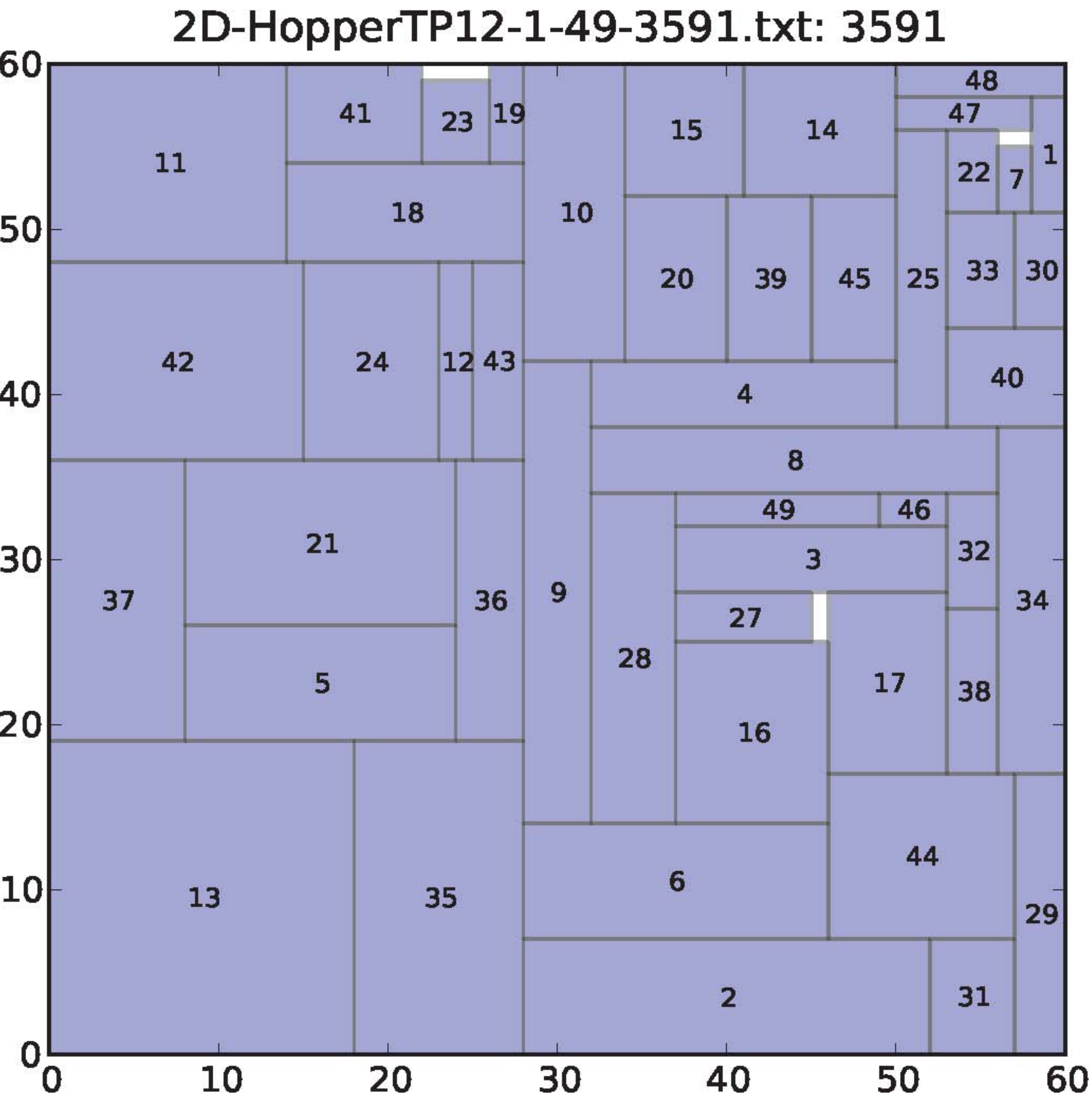


Hopper & Turton, 2001

Instance 4-2 60 x 60

Value: 3591

Previous best: 3580 by a Tabu Search heuristic (Alvarez-Valdes et al., 2007)



Hopper & Turton, 2001

Instance 4-2 60 x 60

Value: 3591

New best known solution!

Previous best: 3580 by a Tabu Search heuristic (Alvarez-Valdes et al., 2007)

BRKGA for constrained 2-dim orthogonal packing

Encoding

- Solutions are encoded as vectors K of

$$2N' = 2 \{ Q[1] + Q[2] + \cdots + Q[N] \}$$

random keys, where Q[i] is the maximum number of rectangles of type i (for i = 1, ..., N) that can be packed.

- K = (k[1], ..., k[N'], k[N'+1], ..., k[2N'])

Encoding

- Solutions are encoded as vectors K of

$$2N' = 2 \{ Q[1] + Q[2] + \cdots + Q[N] \}$$

random keys, where Q[i] is the maximum number of rectangles of type i (for i = 1, ..., N) that can be packed.

- K = (k[1], ..., k[N'], k[N'+1], ..., k[2N'])

Rectangle type packing sequence (RTPS)

Encoding

- Solutions are encoded as vectors K of

$$2N' = 2 \{ Q[1] + Q[2] + \cdots + Q[N] \}$$

random keys, where Q[i] is the maximum number of rectangles of type i (for i = 1, ..., N) that can be packed.

- K = (k[1], ..., k[N'], k[N'+1], ..., k[2N'])

Rectangle type packing sequence (RTPS) — k[1], ..., k[N']

Vector of placement procedures (VPP) — k[N'+1], ..., k[2N']

Decoding

- Simple heuristic to pack rectangles:
  - Make Q[i] copies of rectangle i, for i = 1, ..., N.
  - Order the N' = Q[1] + Q[2] + ⋯ + Q[N] rectangles in some way.
  - Process the rectangles in the above order. Place the rectangle in the stock rectangle according to one of the following heuristics: bottom-left (BL) or left-bottom (LB). If rectangle cannot be positioned, discard it and go on to the next rectangle in the order.

Decoding

- Simple heuristic to pack rectangles:
  - Make Q[i] copies of rectangle i, for i = 1, ..., N.
  - Order the N' = Q[1] + Q[2] + ··· + Q[N] rectangles in some way. **Sort first N' keys to obtain order.**
  - Process the rectangles in the above order. Place the rectangle in the stock rectangle according to one of the following heuristics: bottom-left (BL) or left-bottom (LB). If rectangle cannot be positioned, discard it and go on to the next rectangle in the order.

Decoding

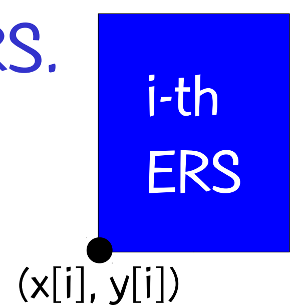
- Simple heuristic to pack rectangles:
  - Make Q[i] copies of rectangle i, for i = 1, ..., N.
  - Order the N' = Q[1] + Q[2] + ⋯ + Q[N] rectangles in some way. **Sort first N' keys to obtain order.**
  - Process the rectangles in the above order. Place the rectangle in the stock rectangle according to one of the following heuristics: bottom-left (BL) or left-bottom (LB). If rectangle cannot be positioned, discard it and go on to the next rectangle in the order. **Use the last N' keys to determine which heuristic to use. If k[N'+i] > 0.5 use LB, else use BL.**

Decoding

- A maximal empty rectangular space (ERS) is an empty rectangular space not contained in any other ERS.
- ERSs are generated and updated using the Difference Process of Lai and Chan (1997).
- When placing a rectangle, we limit ourselves only to maximal ERSs. We order all the maximal ERSs and place the rectangle in the first maximal ERS in which it fits.
- Let (x[i], y[i]) be the coordinates of the bottom left corner of the i-th ERS.

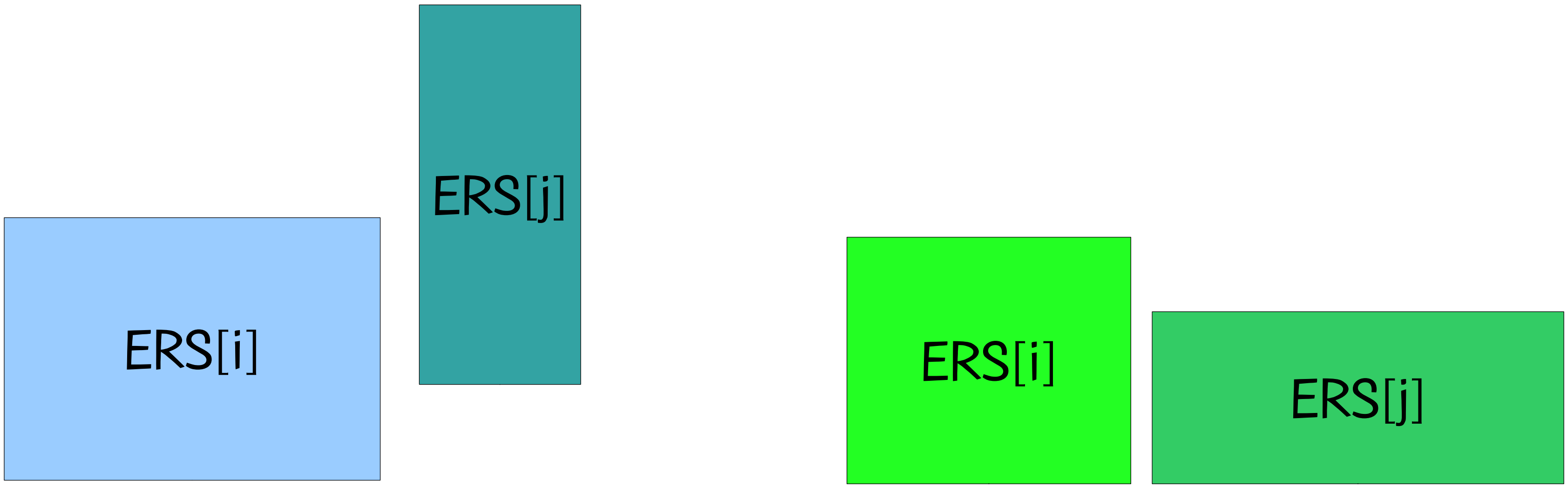
Decoding

- A maximal empty rectangular space (ERS) is an empty rectangular space not contained in any other ERS.
- ERSs are generated and updated using the Difference Process of Lai and Chan (1997).
- When placing a rectangle, we limit ourselves only to maximal ERSs. We order all the maximal ERSs and place the rectangle in the first maximal ERS in which it fits.
- Let (x[i], y[i]) be the coordinates of the bottom left corner of the i-th ERS.

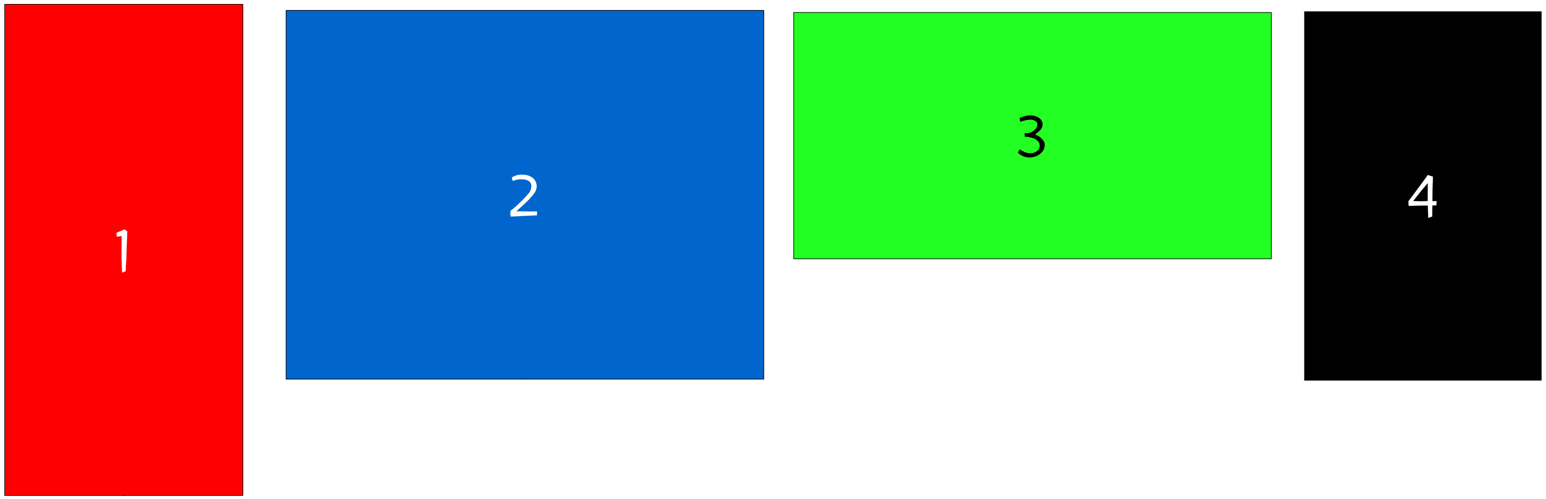


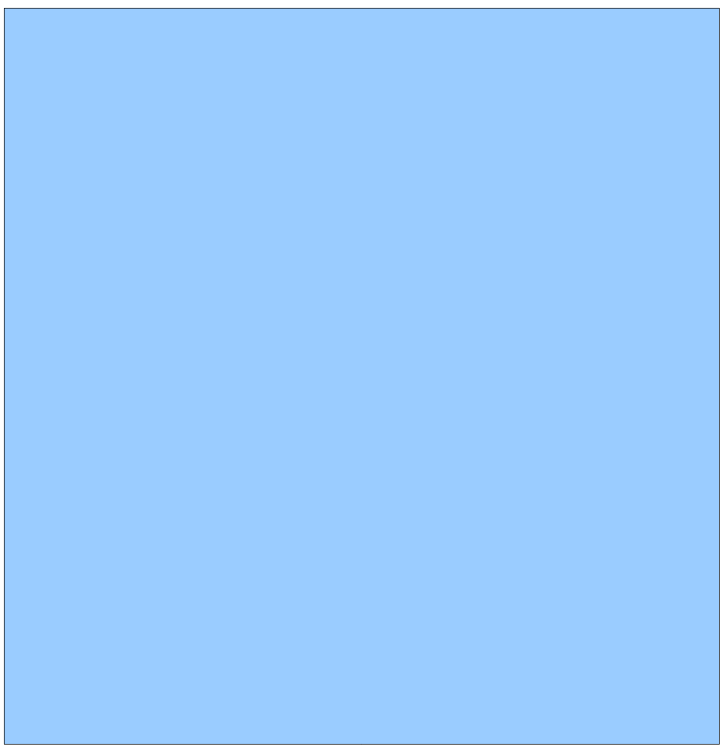
Decoding

- If BL is used, ERSs are ordered such that ERS[i] < ERS[j] if y[i] < y[j] or y[i] = y[j] and x[i] < x[j].



ERS[i] < ERS[j]

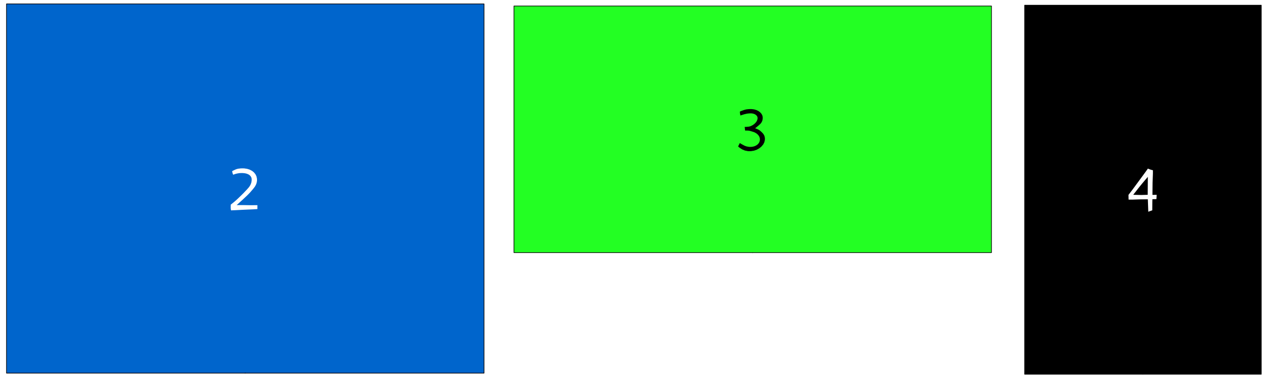


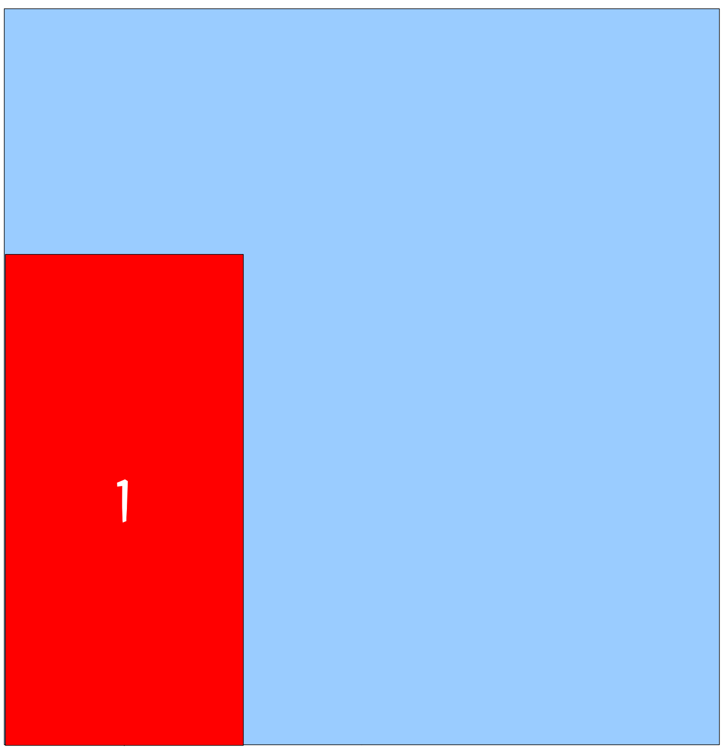


BL can run into problems even on small instances (Liu & Teng, 1999).

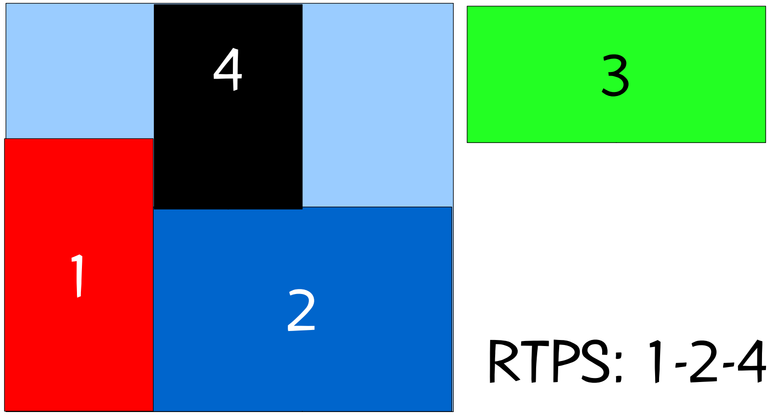
Consider this instance with 4 rectangles.

BL cannot find the optimal solution for any RTPS.

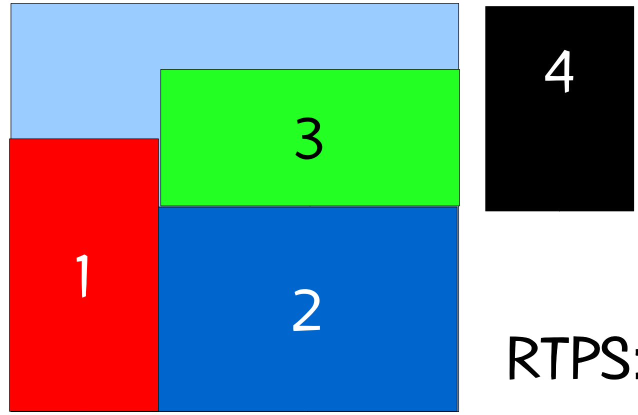




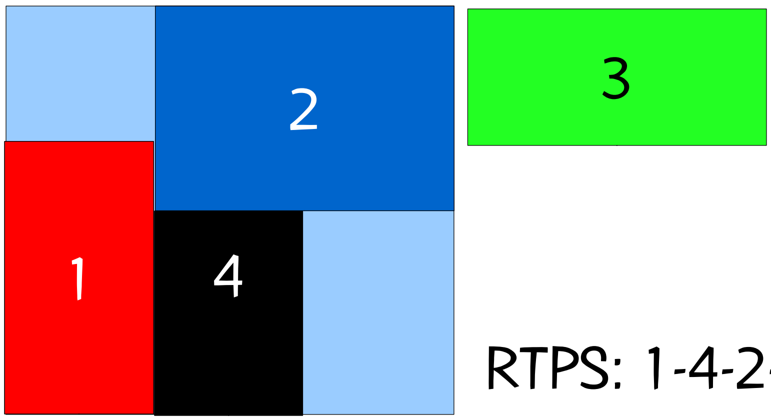
We show 6 rectangle type packing sequences (RTPS's) where we fix rectangle 1 in the first position.



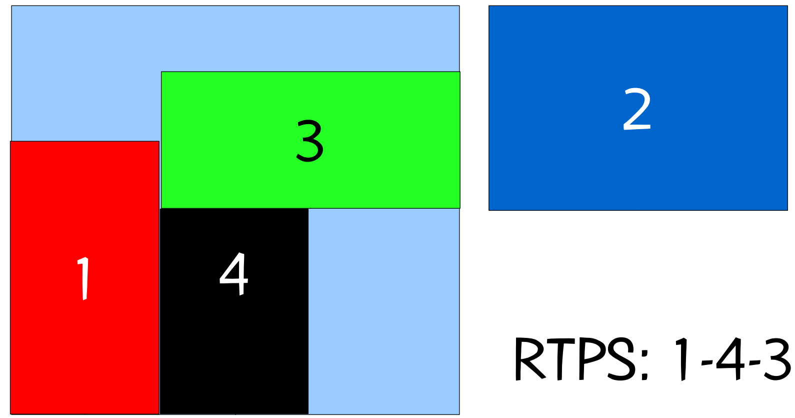
RTPS: 1-2-4-3



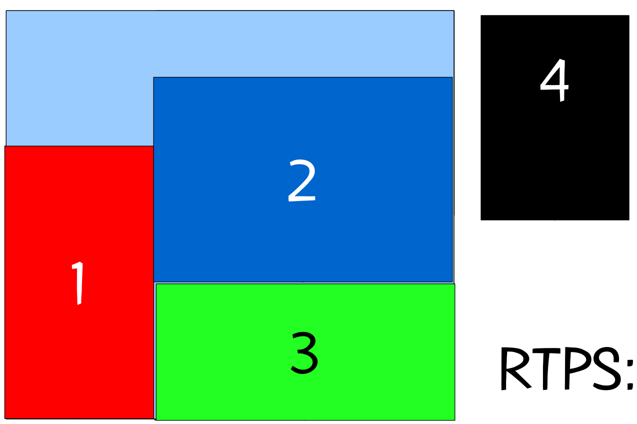
RTPS: 1-2-3-4



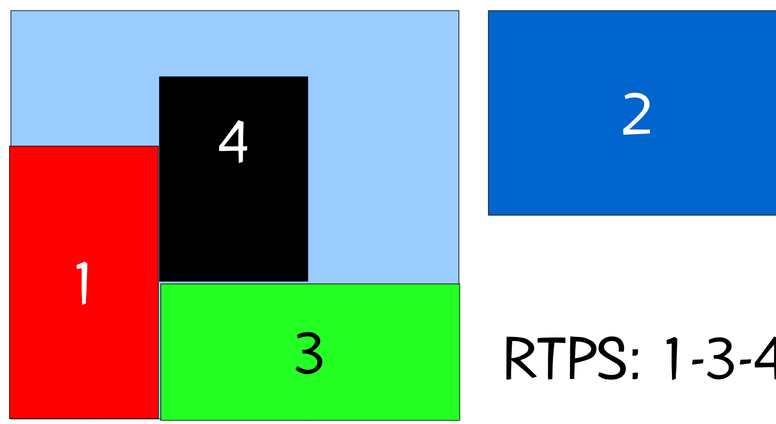
RTPS: 1-4-2-3



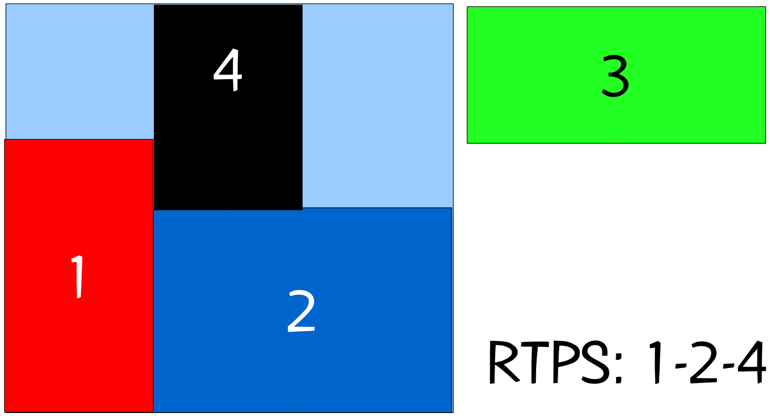
RTPS: 1-4-3-2



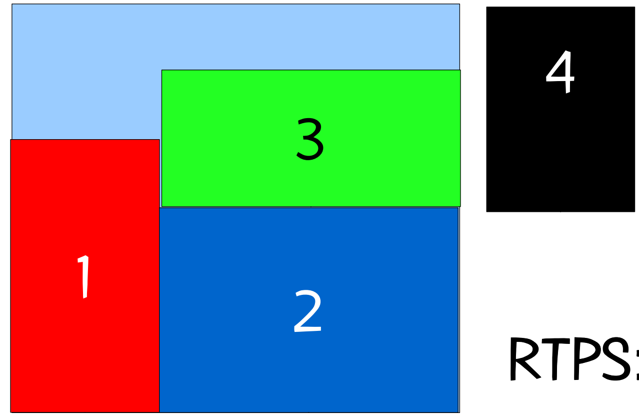
RTPS: 1-3-2-4



RTPS: 1-3-4-2

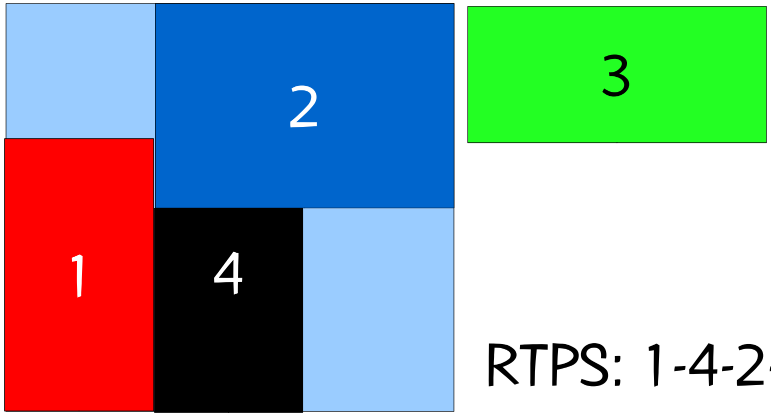


RTPS: 1-2-4-3

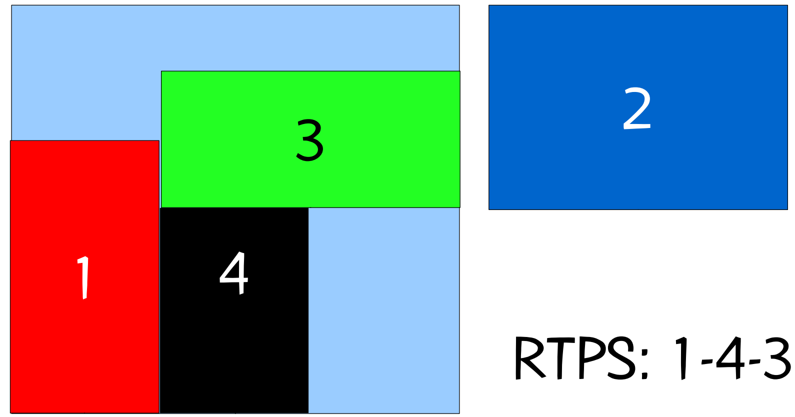


Similar infeasibilities are observed if 2, 3, or 4 is the first rectangle in the RTPS.

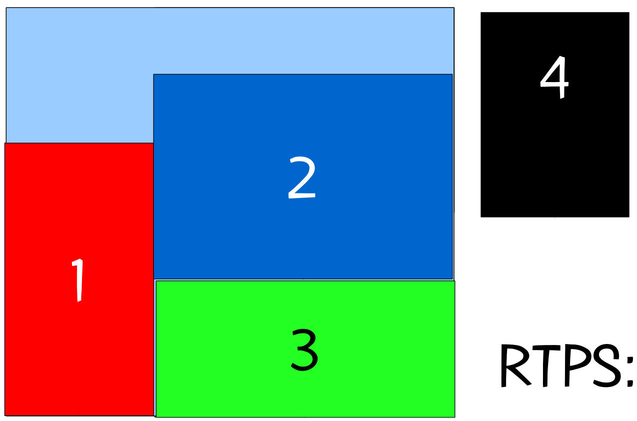
RTPS: 1-2-3-4



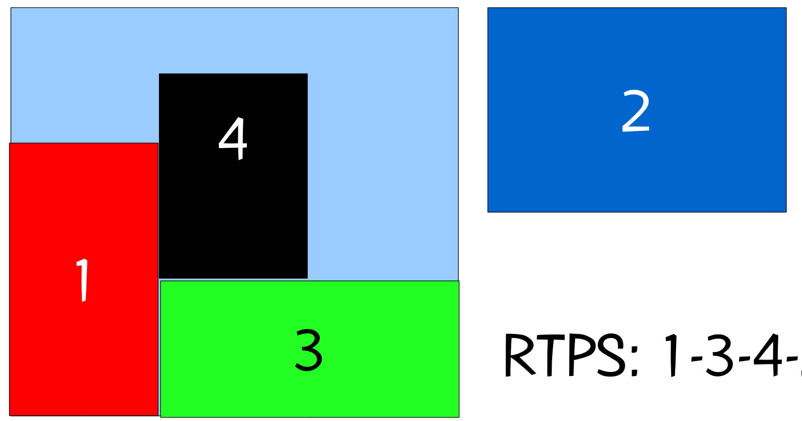
RTPS: 1-4-2-3



RTPS: 1-4-3-2



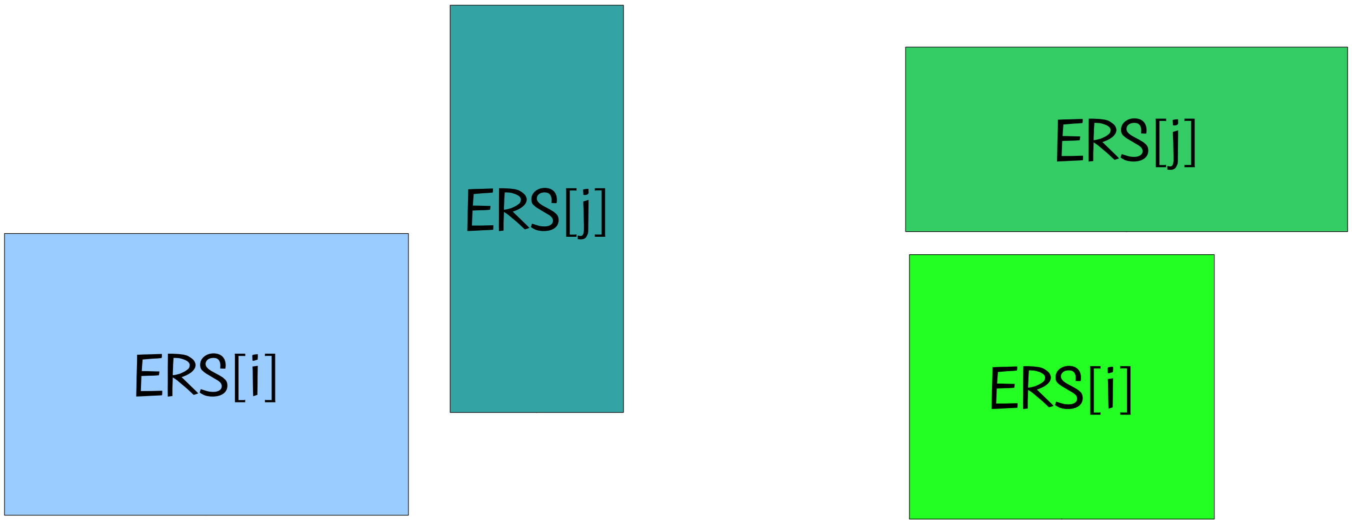
RTPS: 1-3-2-4



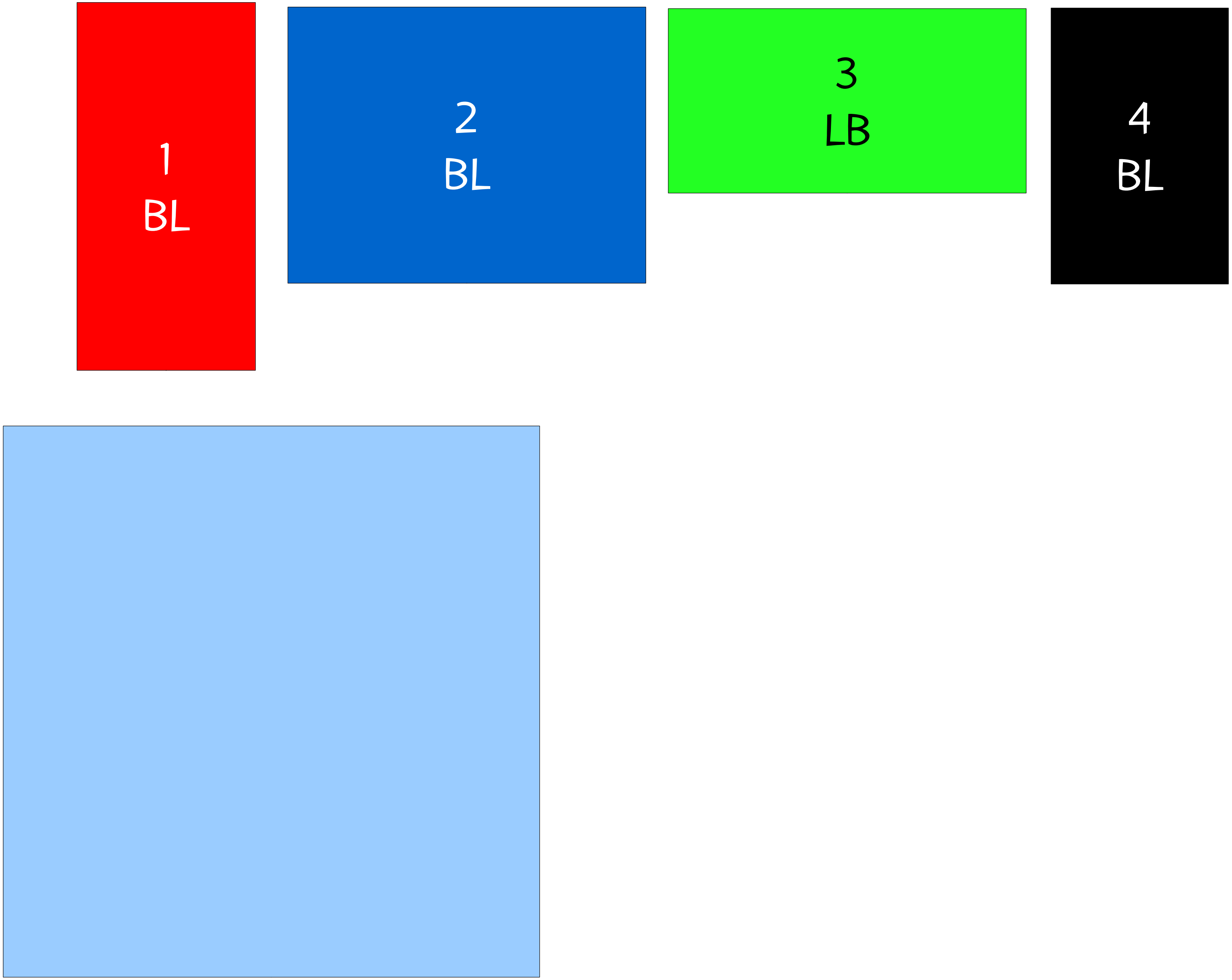
RTPS: 1-3-4-2

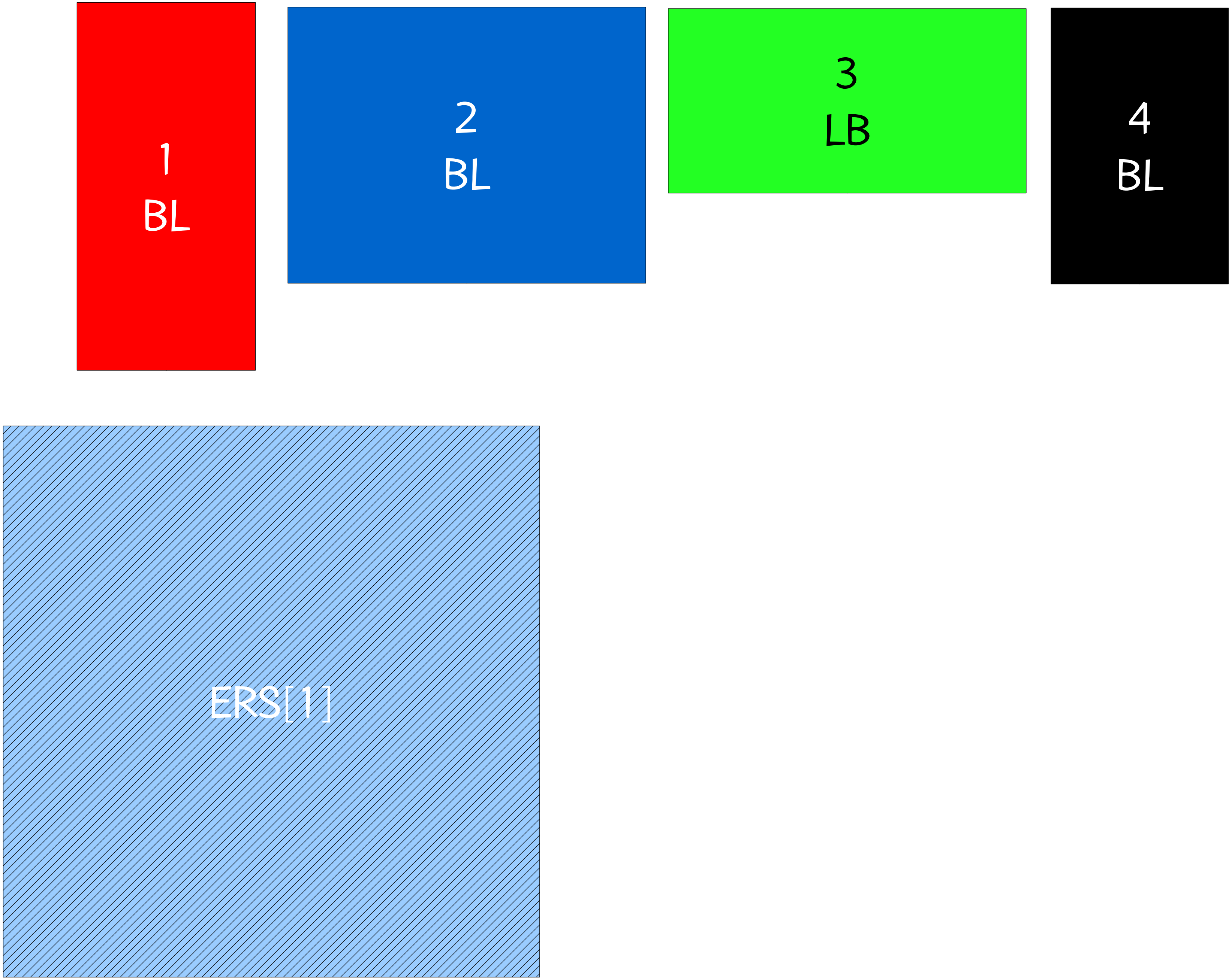
Decoding

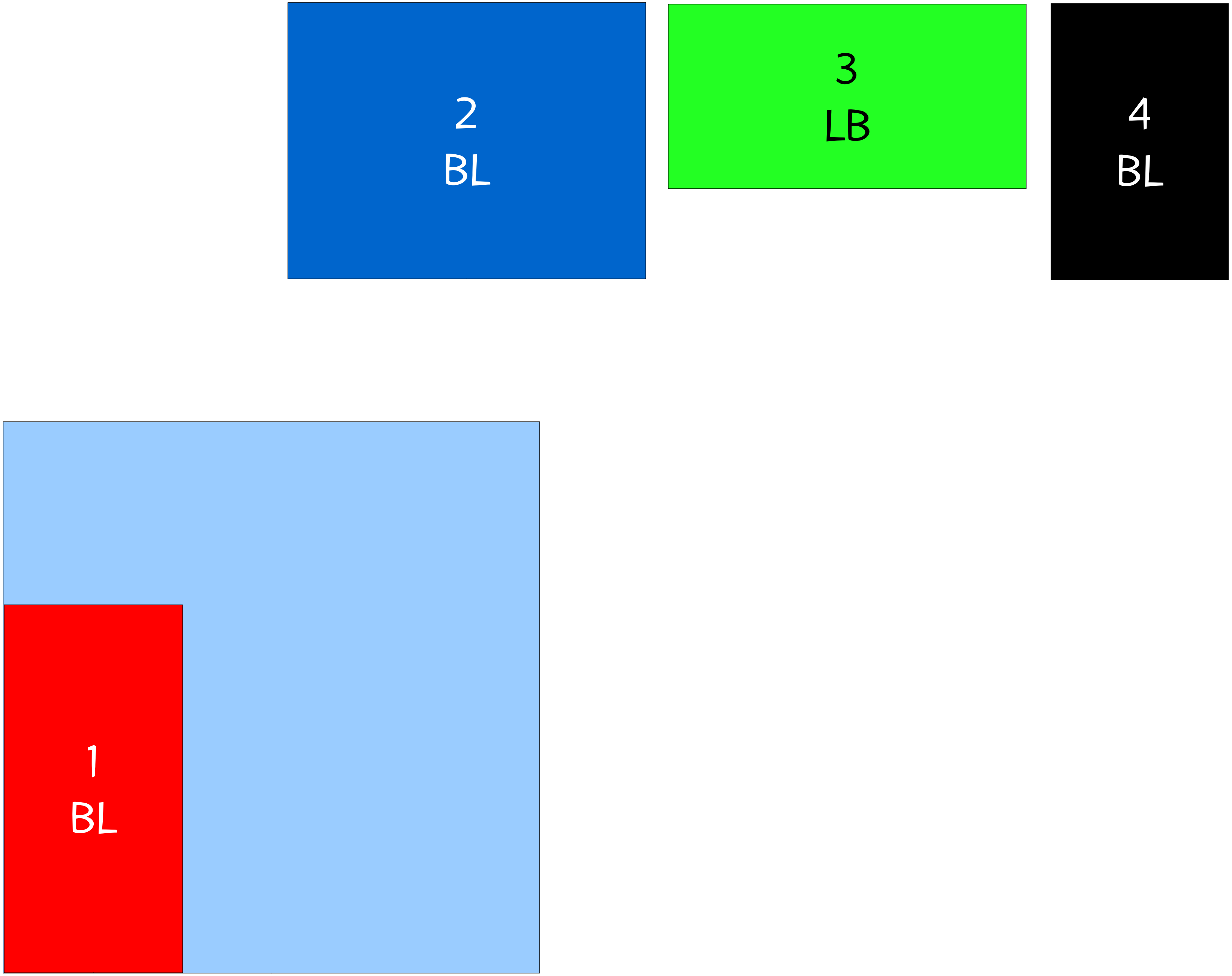
- If LB is used, ERSs are ordered such that ERS[i] < ERS[j] if x[i] < x[j] or x[i] = x[j] and y[i] < y[j].

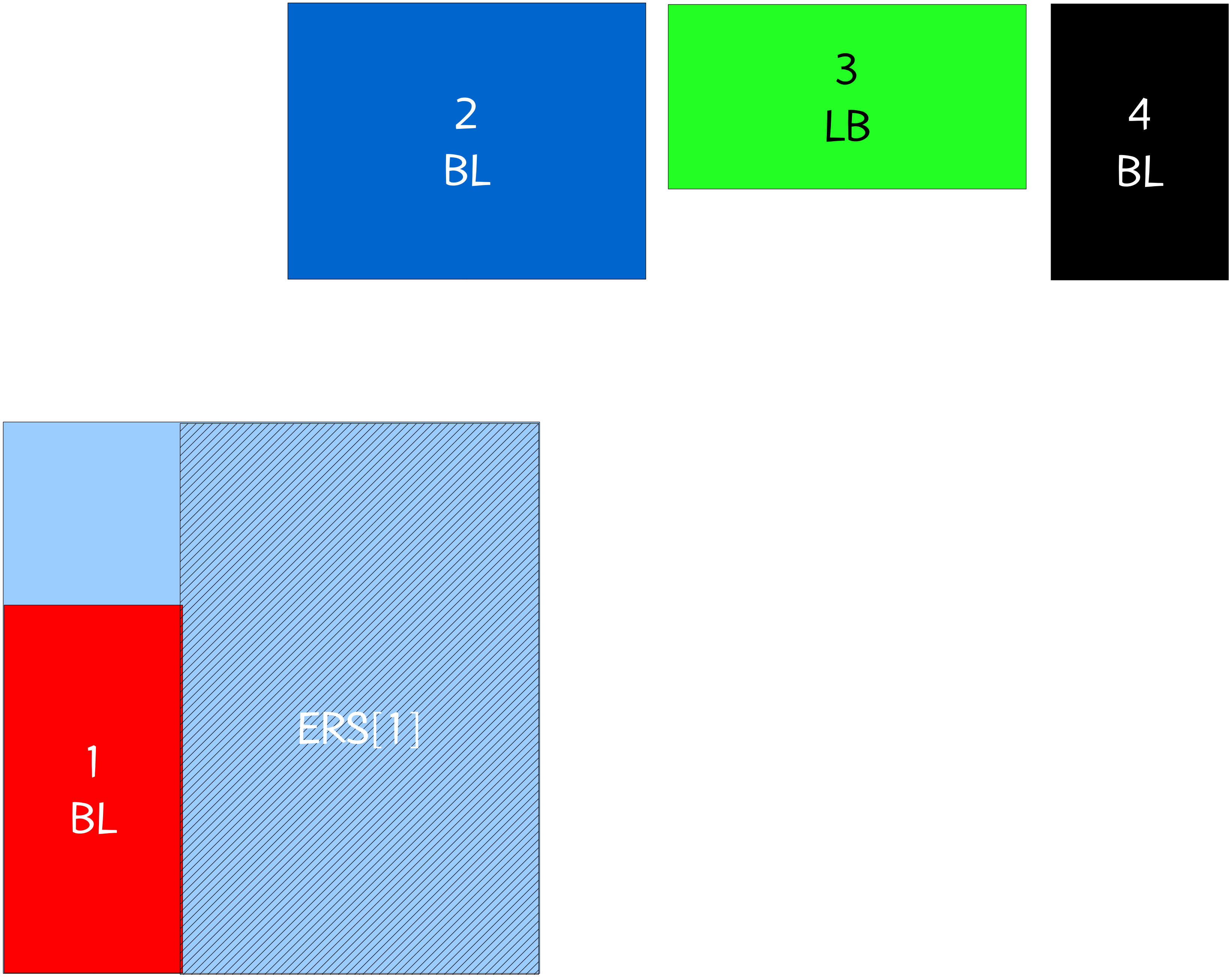


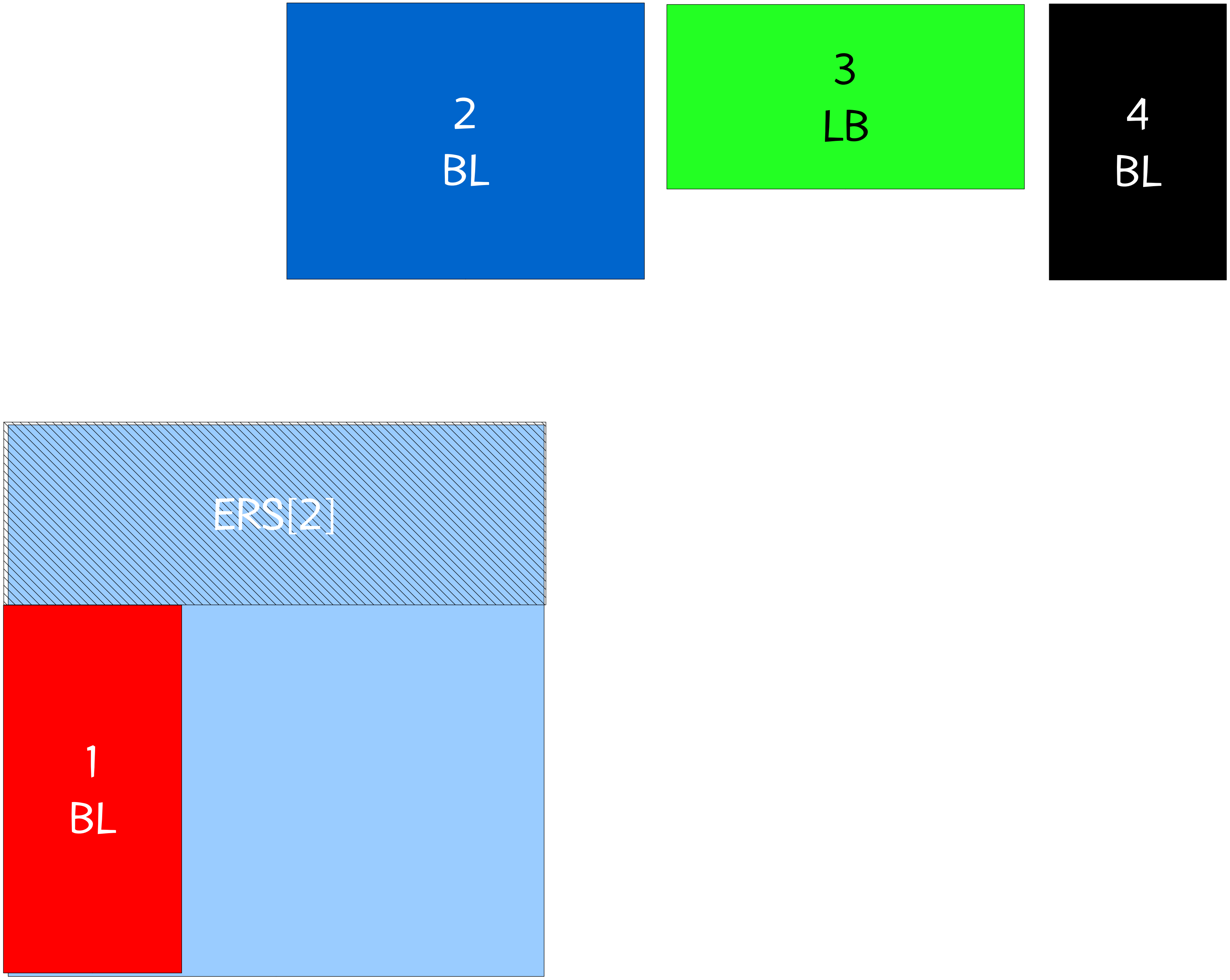
ERS[i] < ERS[j]

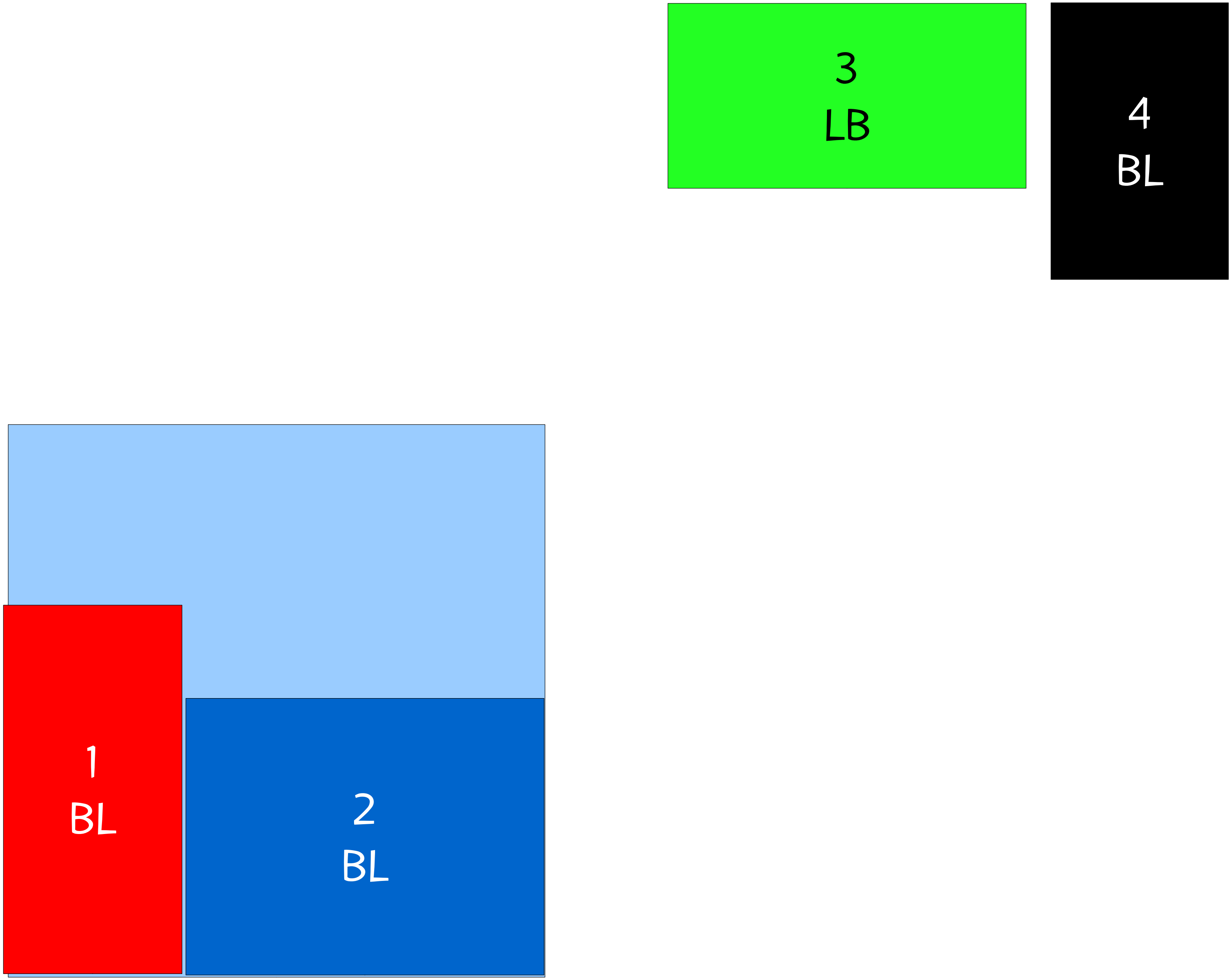

1
BL
2
BL
3
LB
4
BL

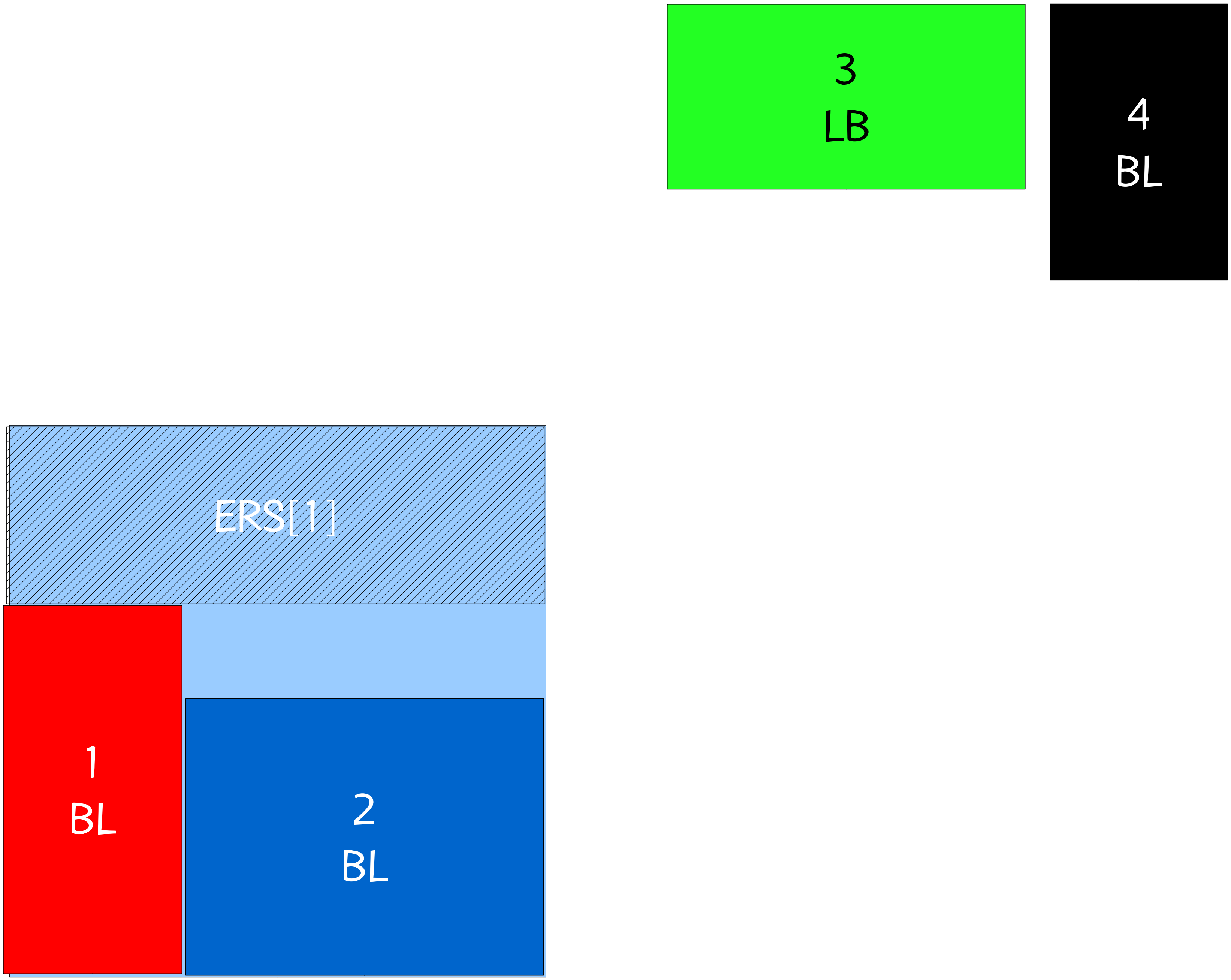

1
BL
2
BL
3
LB
4
BL
ERS[1]

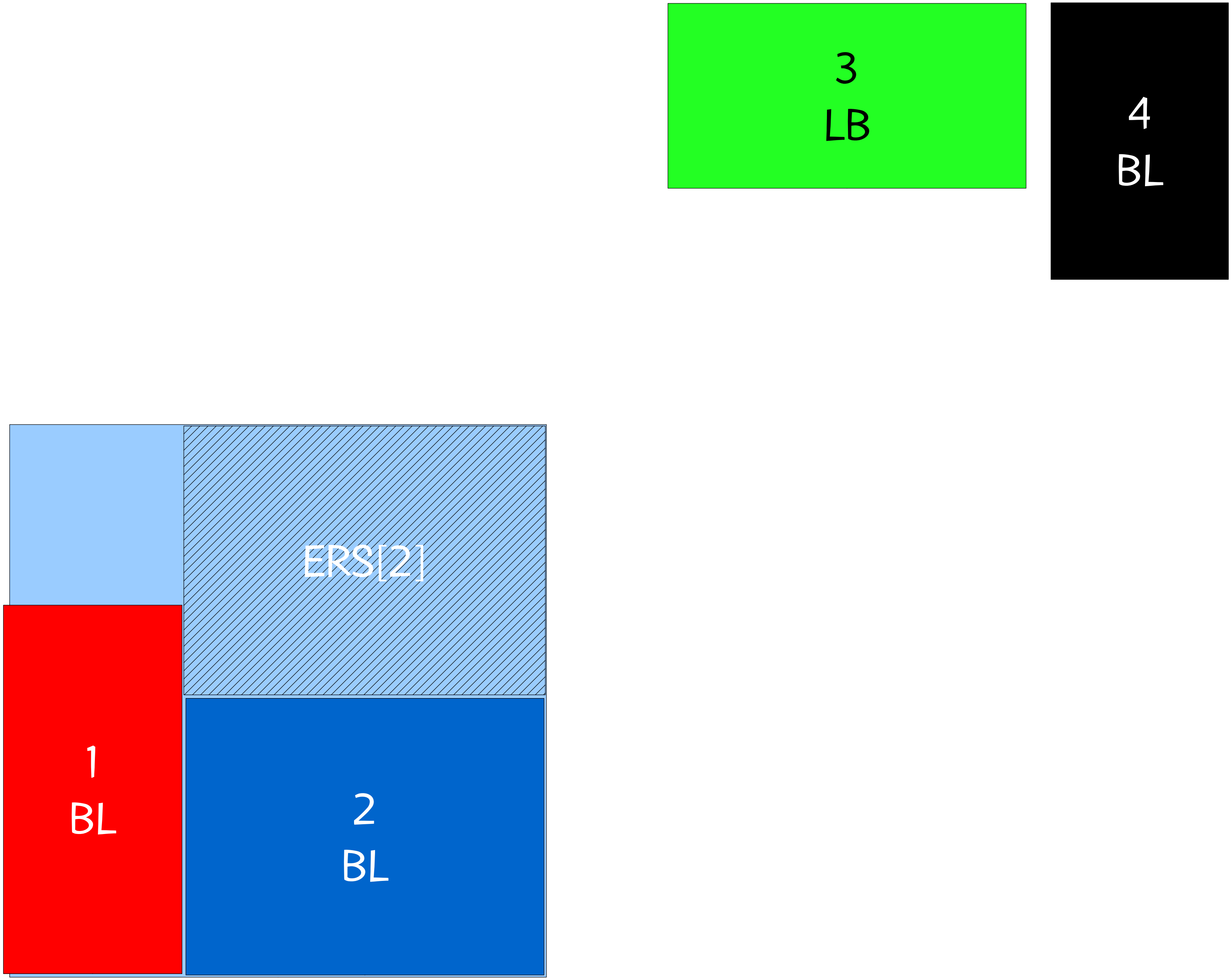

2
BL
3
LB
4
BL
1
BL

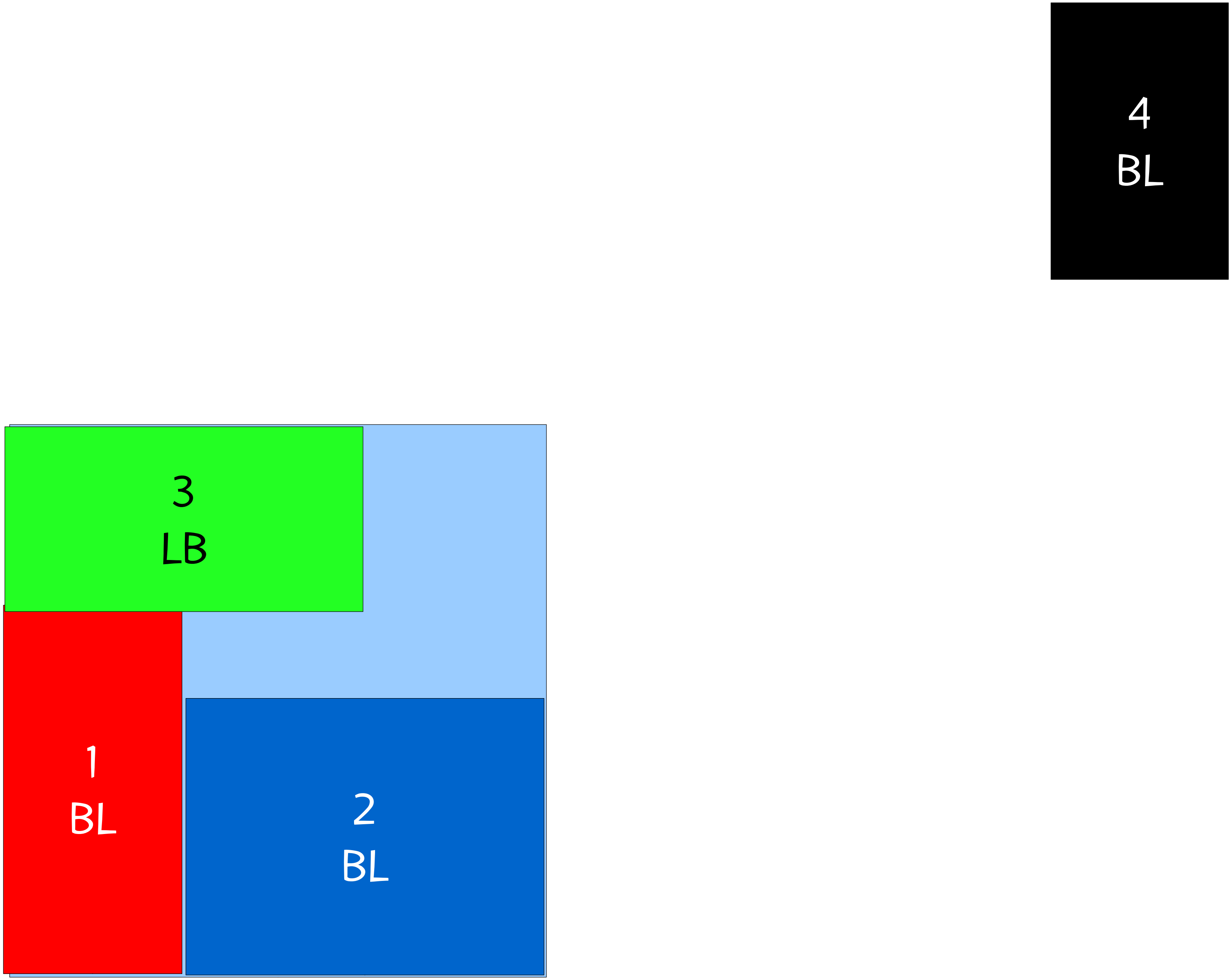

2
BL
3
LB
4
BL
1
BL
ERS[1]

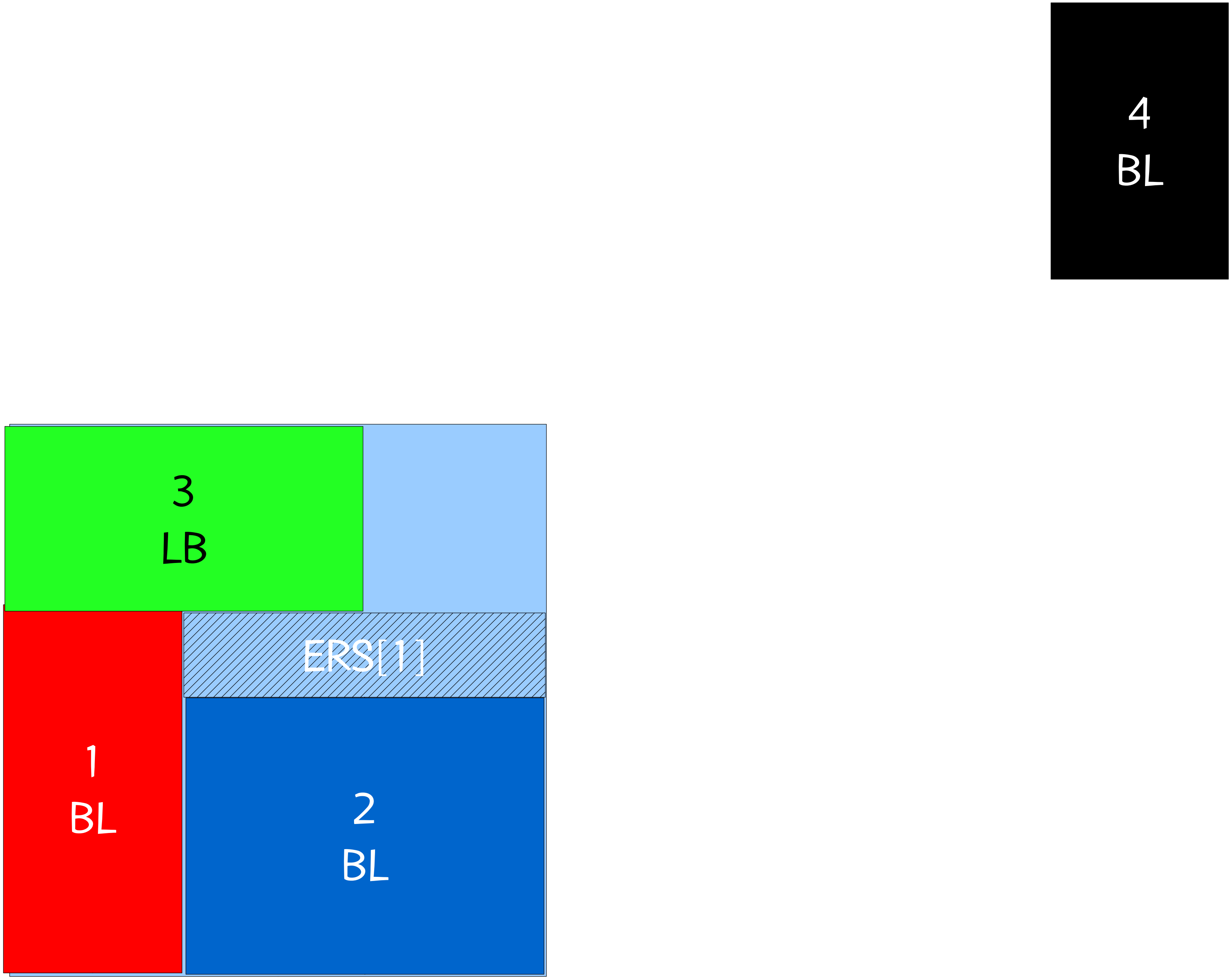

2
BL
3
LB
4
BL
ERS[2]
1
BL

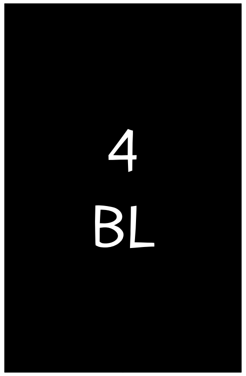

3
LB
4
BL
1
BL
2
BL


3
LB
4
BL
ERS[1]
1
BL
2
BL

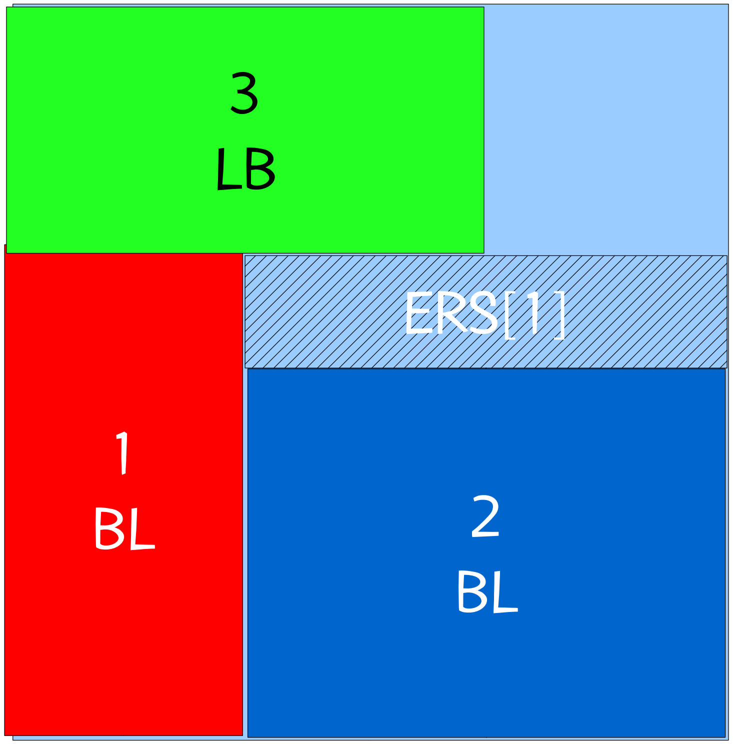

3
LB
4
BL
ERS[2]
1
BL
2
BL

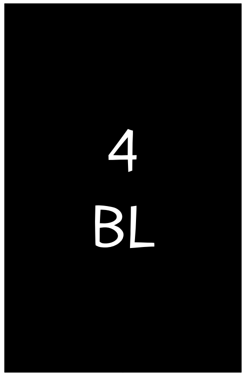

4
BL
3
LB
1
BL
2
BL


4
BL
3
LB
ERS[1]
1
BL
2
BL

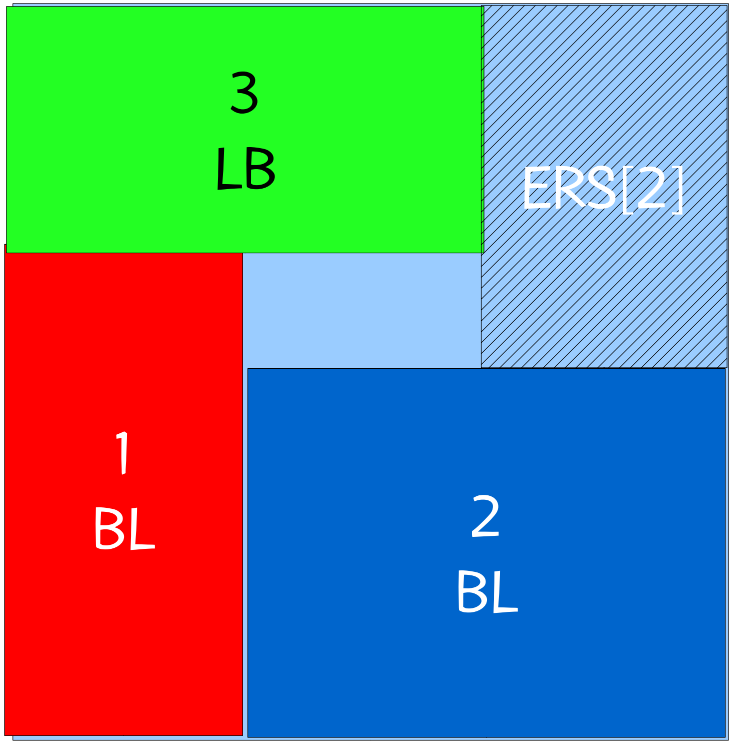


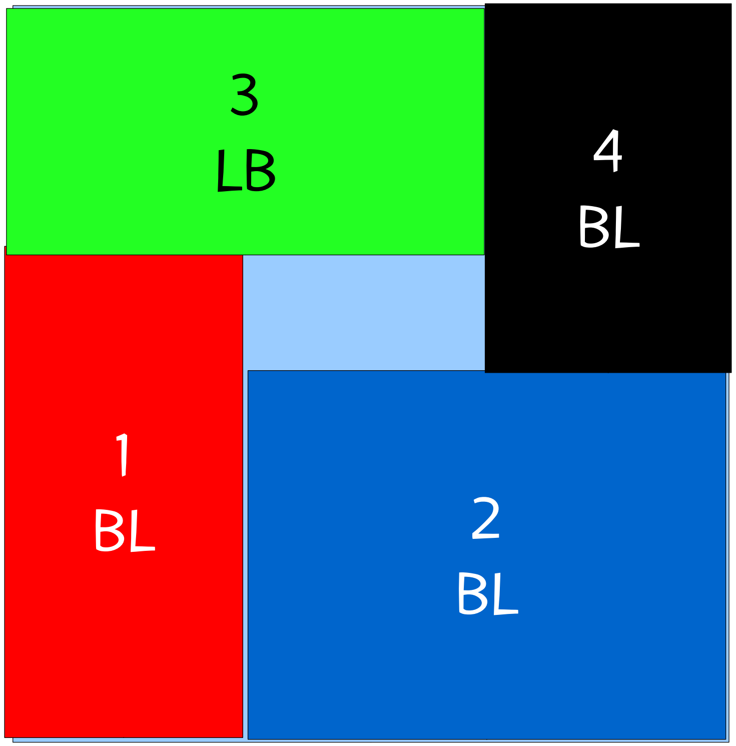
4 does not fit in ERS[1].





4 does fit in ERS[2].





Optimal solution!

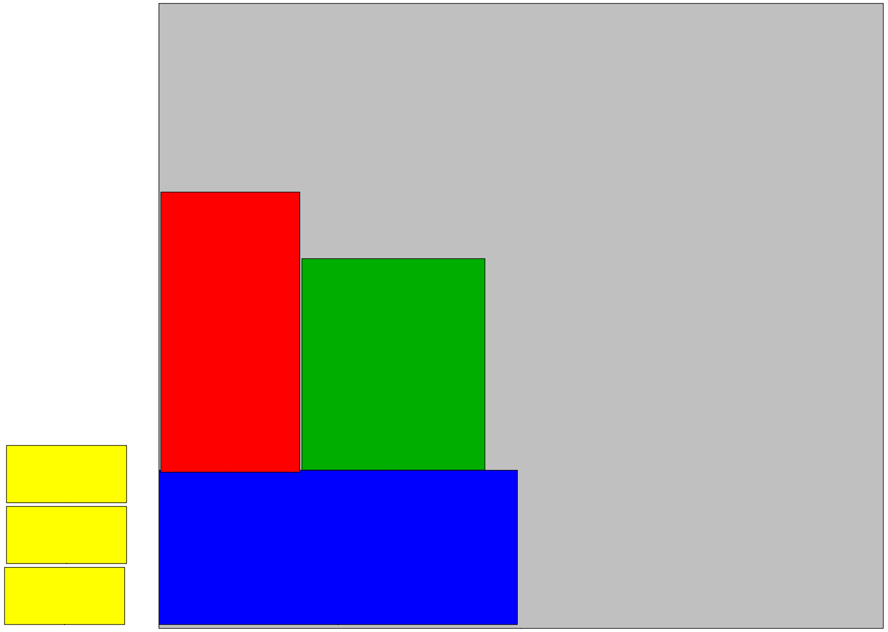
Implementation details

Packing layers

- When placing a rectangle type in an ERS we try to build a layer containing several rectangles of that rectangle type.
- We use two types of layers:
  - Horizontal layer when using BL
  - Vertical layer when using LB

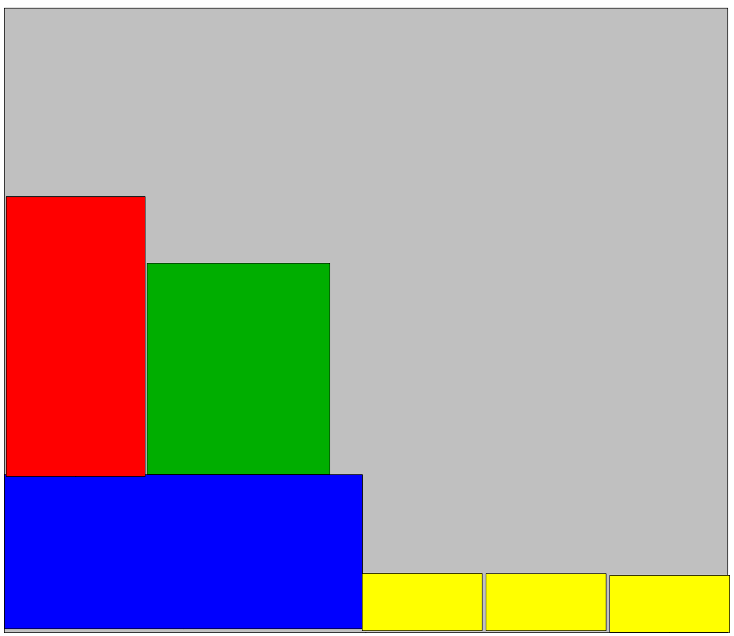
Packing layers

Horizontal layer (BL)



Packing layers

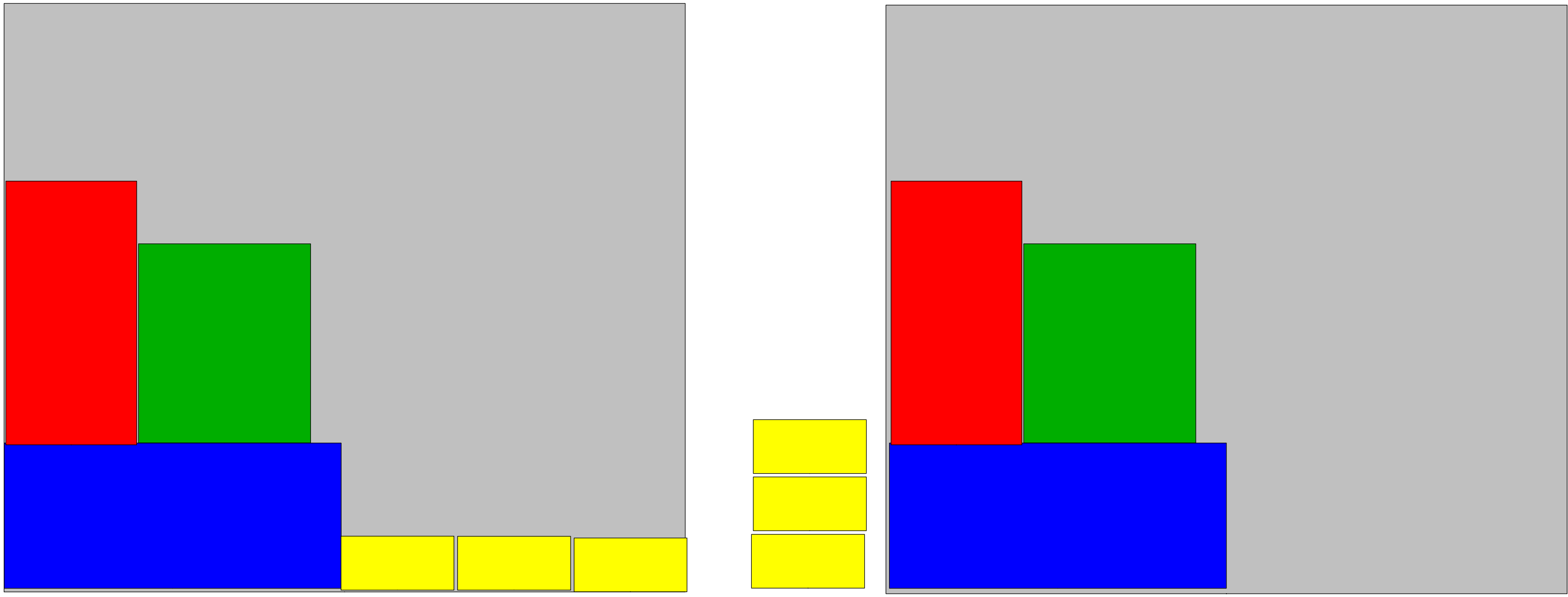
Horizontal layer (BL)



Packing layers

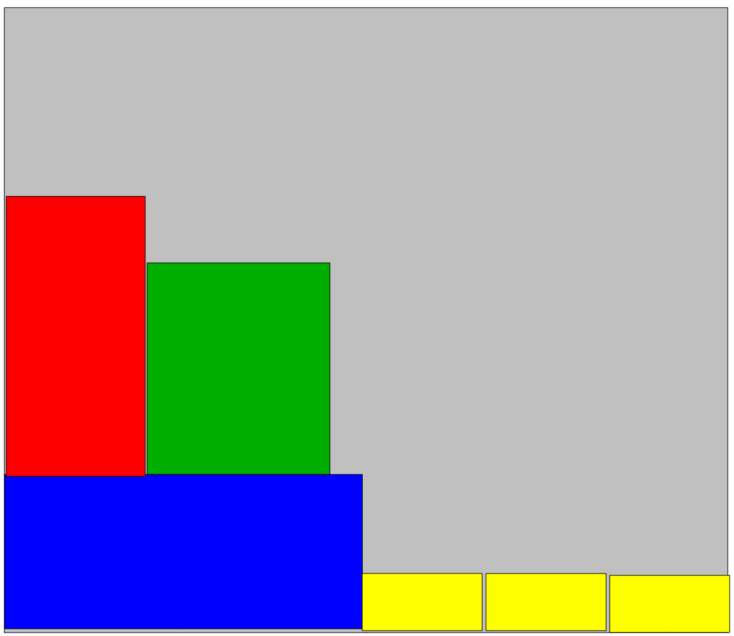
Horizontal layer (BL)

Vertical layer (LB)

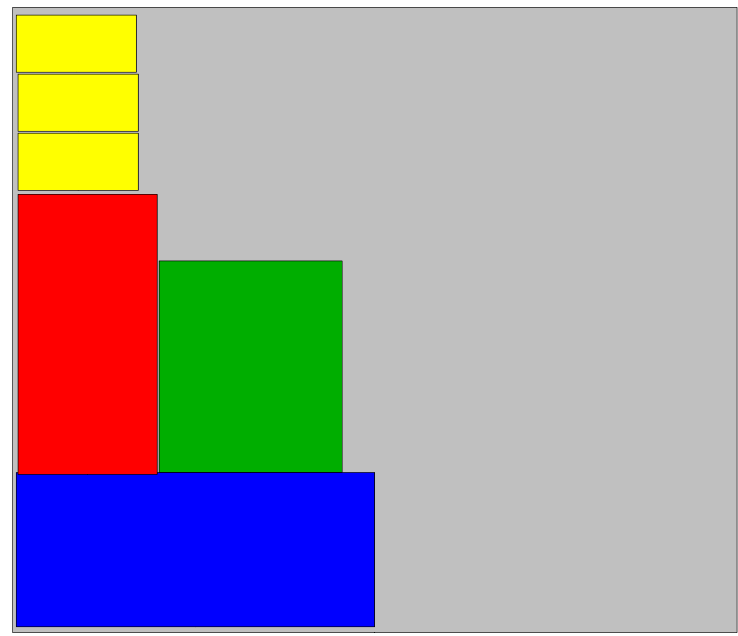


Packing layers

Horizontal layer (BL)



Vertical layer (LB)



Population initialization

- Initial population does not consist entirely of random vectors.
- Four non-random vectors are introduced into each population.
- The chromosomes of these four solutions are generated such that their rectangle type packing sequences (RTPSes) are equivalent to packing rectangles in decreasing order of their values. Four variations of the placement procedure are considered:
  – Random, all BL, all LB, alternating between BL and LB

Modified total value fitness function

- Natural fitness function is $v[1]\, r[1] + v[2]\, r[2] + \cdots + v[N]\, r[N]$ where $r[i]$ is the number of rectangles of type i to be packed and $v[i]$ is the value of a rectangle of type i.
- Two solution may have the same natural fitness but one may be more "fit" than the other.
- We use an adaptation of the modified measure proposed by Gonçalves (2007) that is able to capture the improvement potential of different packings with identical natural fitness function values.

Modified total value fitness function

Modified total value fitness function is

$$v[1]\, r[1] + v[2]\, r[2] + \cdots + v[N]\, r[N] +$$

$$\frac{0.03 \times \text{min } v[i] \text{ of all rectangles}}{\text{area of stock rectangle}} \times \textbf{area largest ERS left over}$$

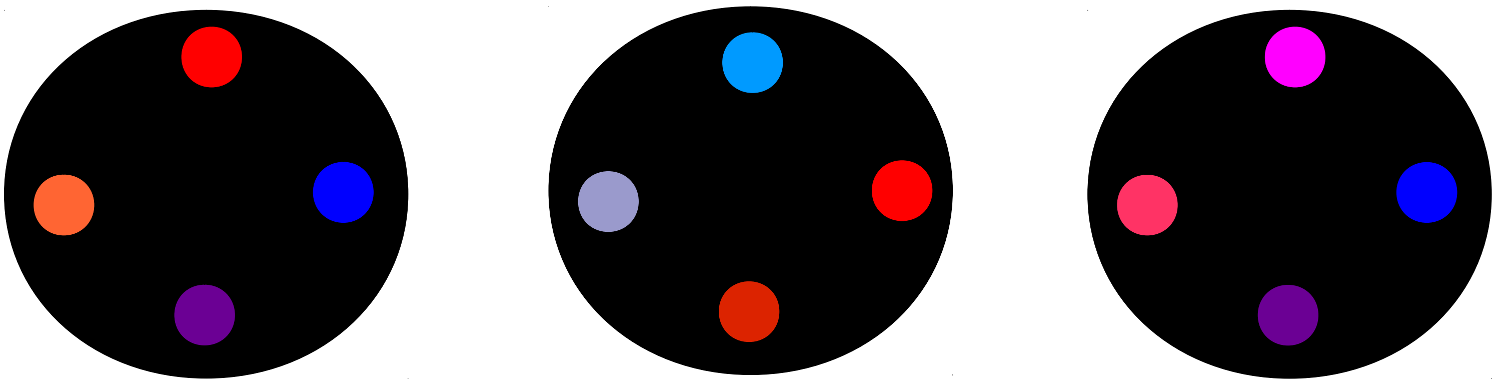
Ties are broken by area of largest maximal empty rectangular space (ERS) left over.

Handling lower bounds

To handle the lower bounds P[i] on r[i] we impose a penalty of 10^{10} which is subtracted from the modified fitness function if r[i] < P[i] for some i = 1, ..., N.

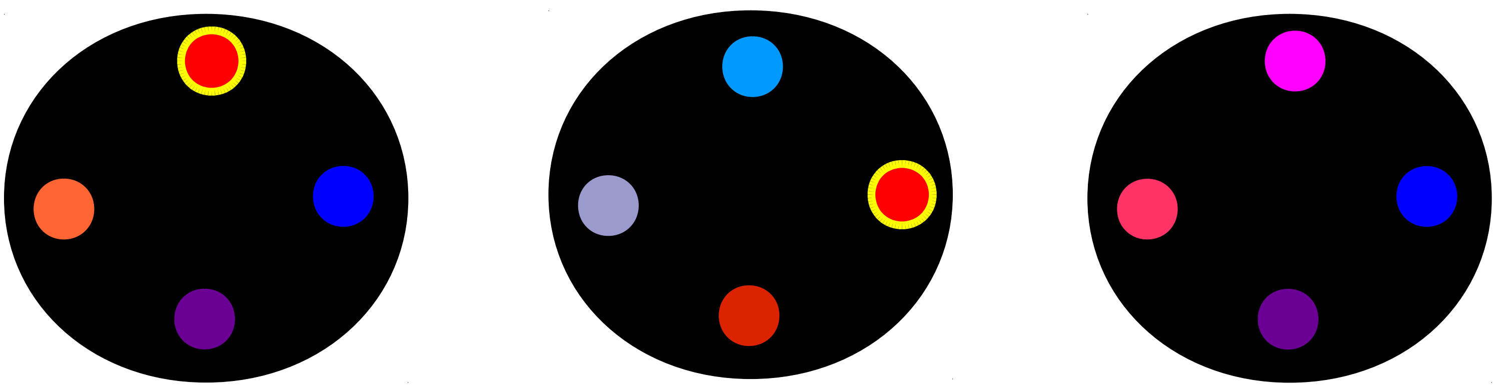
Multi-population strategy

- Three populations are evolved simultaneously.



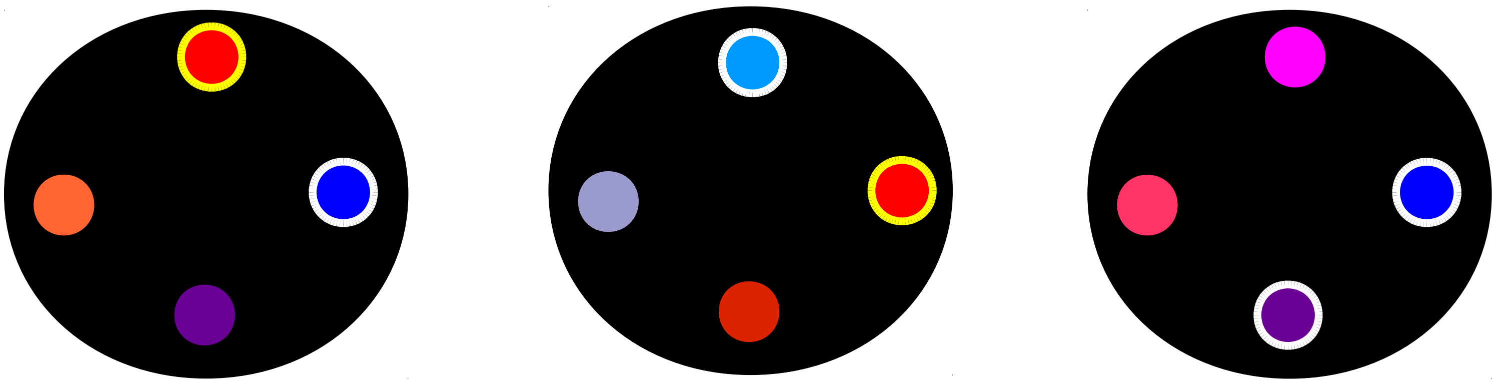
Multi-population strategy

- Three populations are evolved simultaneously.
- Every 15 generations populations exchange information:
  - The best two solutions over all three populations are copied to the populations where they are not present.
  - They replace the worst solution(s) in the population.



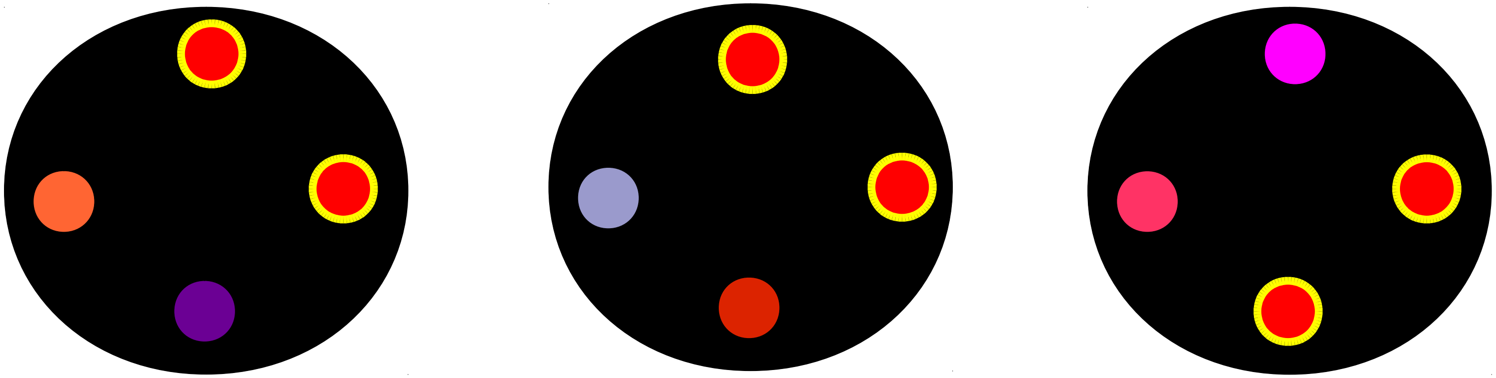
Multi-population strategy

- Three populations are evolved simultaneously.
- Every 15 generations populations exchange information:
  - The best two solutions over all three populations are copied to the populations where they are not present.
  - They replace the worst solution(s) in the population.



Multi-population strategy

- Three populations are evolved simultaneously.
- Every 15 generations populations exchange information:
  - The best two solutions over all three populations are copied to the populations where they are not present.
  - They replace the worst solution(s) in the population.



Parallel implementation

- Fitness evaluations are done in parallel.
- Easy to implement using OpenMP in C++.
- In multi-core CPUs results in almost linear speed-ups.
- Experiments done on an Intel 2.66 GHz Xeon Quadcore CPU using the Linux CentOS 5 operating sysem.

Experimental results

Design

- We compare solution values obtained by the parallel multi-population BRKGA with solutions obtained by the heuristics that produced the best computational results to date:

Design

- We compare solution values obtained by the parallel multi-population BRKGA with solutions obtained by the heuristics that produced the best computational results to date:
  - PH: population-based heuristic of Beasley (2004)

Design

- We compare solution values obtained by the parallel multi-population BRKGA with solutions obtained by the heuristics that produced the best computational results to date:
  - PH: population-based heuristic of Beasley (2004)
  - GA: genetic algorithm of Hadjiconsantinou & Iori (2007)

Design

- We compare solution values obtained by the parallel multi-population BRKGA with solutions obtained by the heuristics that produced the best computational results to date:
  - PH: population-based heuristic of Beasley (2004)
  - GA: genetic algorithm of Hadjiconsantinou & Iori (2007)
  - GRASP: greedy randomized adaptive search procedure of Alvarez-Valdes et al. (2005)

Design

- We compare solution values obtained by the parallel multi-population BRKGA with solutions obtained by the heuristics that produced the best computational results to date:
  - PH: population-based heuristic of Beasley (2004)
  - GA: genetic algorithm of Hadjiconsantinou & Iori (2007)
  - GRASP: greedy randomized adaptive search procedure of Alvarez-Valdes et al. (2005)
  - TABU: tabu search of Alvarez-Valdes et al. (2007)

Design

- We use the same set of test problems considered by Alvarez-Valdes et al. (2007):
  - 21 instances with known optimal solutions from the literature {Beasley (1985), Hadjiconstantinou & Christofides (1995), Wang (1983), Christofides & Whitlock (1977), Fekete & Schepers (2004)};

Design

- We use the same set of test problems considered by Alvarez-Valdes et al. (2007):
  - 630 large problems, randomly generated by Beasley (2004), following Fekete & Schepers (2004);

Design

- We use the same set of test problems considered by Alvarez-Valdes et al. (2007):
  - 31 zero-waste instances used by Lueng et al. (2003);

Design

- We use the same set of test problems considered by Alvarez-Valdes et al. (2007):
  - 21 doubly constrained problems resulting from the introduction of lower bounds for some rectangle types in the first set of Beasley (2004).

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:
  - Elite set: top 25% solutions in population

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:
  - Elite set: top 25% solutions in population
  - Mutants: 15% of population

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:
  - Elite set: top 25% solutions in population
  - Mutants: 15% of population
  - Elite parent inheritance probability: 70%

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:
  - Elite set: top 25% solutions in population
  - Mutants: 15% of population
  - Elite parent inheritance probability: 70%
  - Population size: 15 times the number of rectangles in problem instance (at most 2000 solutions)

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:
  - Elite set: top 25% solutions in population
  - Mutants: 15% of population
  - Elite parent inheritance probability: 70%
  - Population size: 15 times the number of rectangles in problem instance (at most 2000 solutions)
  - Number of populations: 3

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:
  - Elite set: top 25% solutions in population
  - Mutants: 15% of population
  - Elite parent inheritance probability: 70%
  - Population size: 15 times the number of rectangles in problem instance (at most 2000 solutions)
  - Number of populations: 3
  - Exchange best two solutions every 15 generations

Configuration of the BRKGA

- Small pilot study determined the configuration of the BRKGA:
  - Elite set: top 25% solutions in population
  - Mutants: 15% of population
  - Elite parent inheritance probability: 70%
  - Population size: 15 times the number of rectangles in problem instance (at most 2000 solutions)
  - Number of populations: 3
  - Exchange best two solutions every 15 generations
  - Stop after 1000 generations

Overall average percentage deviation from optimal/best lower bound with 4 variant

Set	Description	BL	BL-L	BL-LB-L	BL-LB-L-4NR
1	From literature (optimal)	0.00	0.00	0.00	0.00
2	Large random	1.04	1.00	0.87	0.83
3	Zero-waste	0.48	0.48	0.24	0.17
4	Doubly constrained	6.36	6.36	6.36	6.36

BL: Using only Bottom Left placement
BL-L: BL with layers
BL-LB-L: BL and Left Bottom with layers
BL-LB-L-4NR: BL-LB-L with four non-random starting solutions

Overall average percentage deviation from optimal/best lower bound

Problem	PH	GA	GRASP	TABU	BRKGA BL-LB-L-4NR
From literature (optimal)	5.49	**0.00**	0.19	**0.00**	**0.00**
Large random	1.67	1.32	1.07	0.98	**0.83**
Zero-waste			1.68	0.42	**0.17**
Doubly constrained	8.11		7.36	6.62	**6.36**

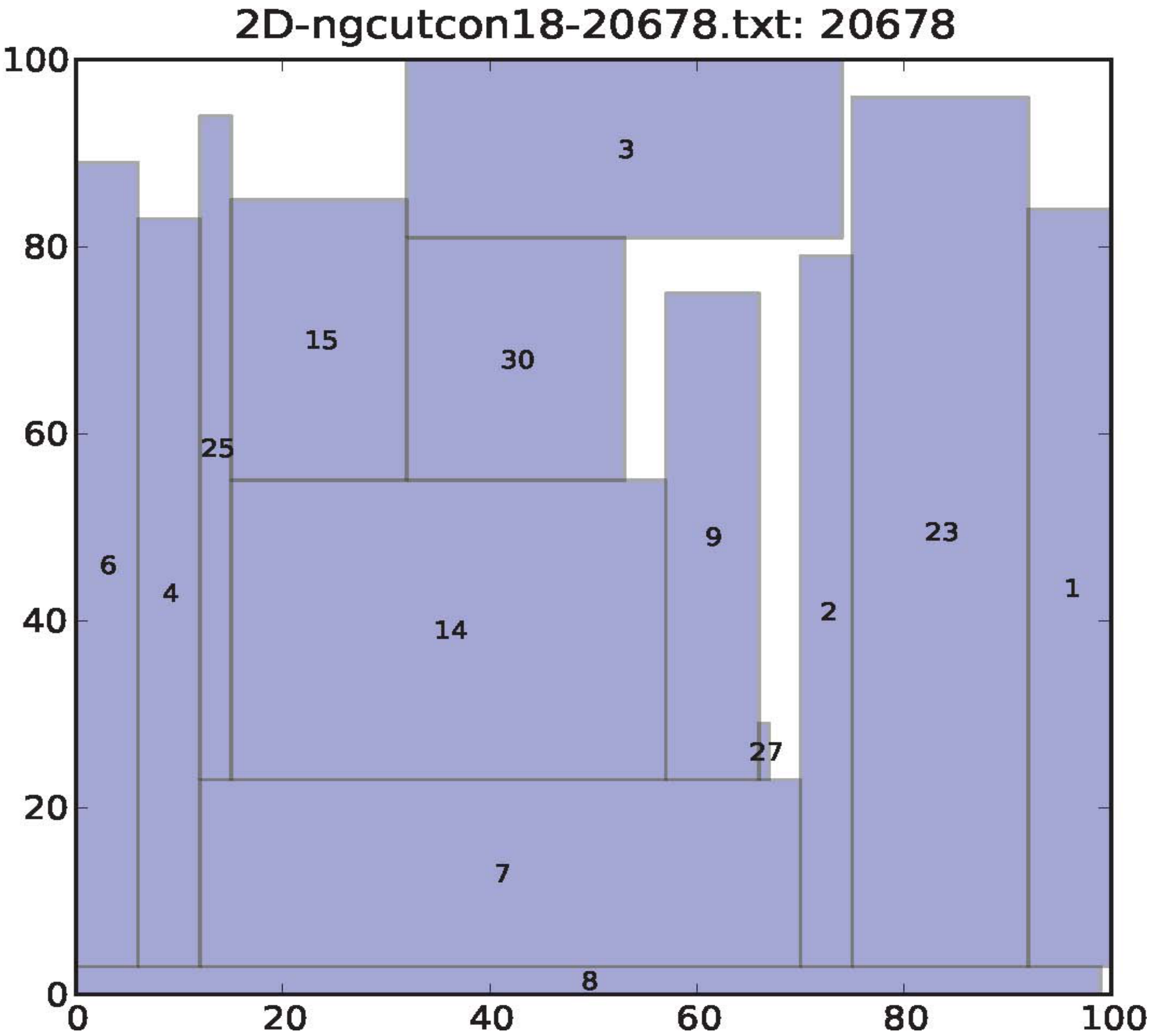
Number of best solutions / total instances

Problem	PH	GA	GRASP	TABU	BRKGA BL-LB-L-4NR
From literature (optimal)	13/21	**21/21**	18/21	**21/21**	**21/21**
Large random*	0/21	0/21	5/21	8/21	**20/21**
Zero-waste			5/31	17/31	**30/31**
Doubly constrained	11/21		12/21	17/21	**19/21**

* For large random: number of best average solutions / total instance classes

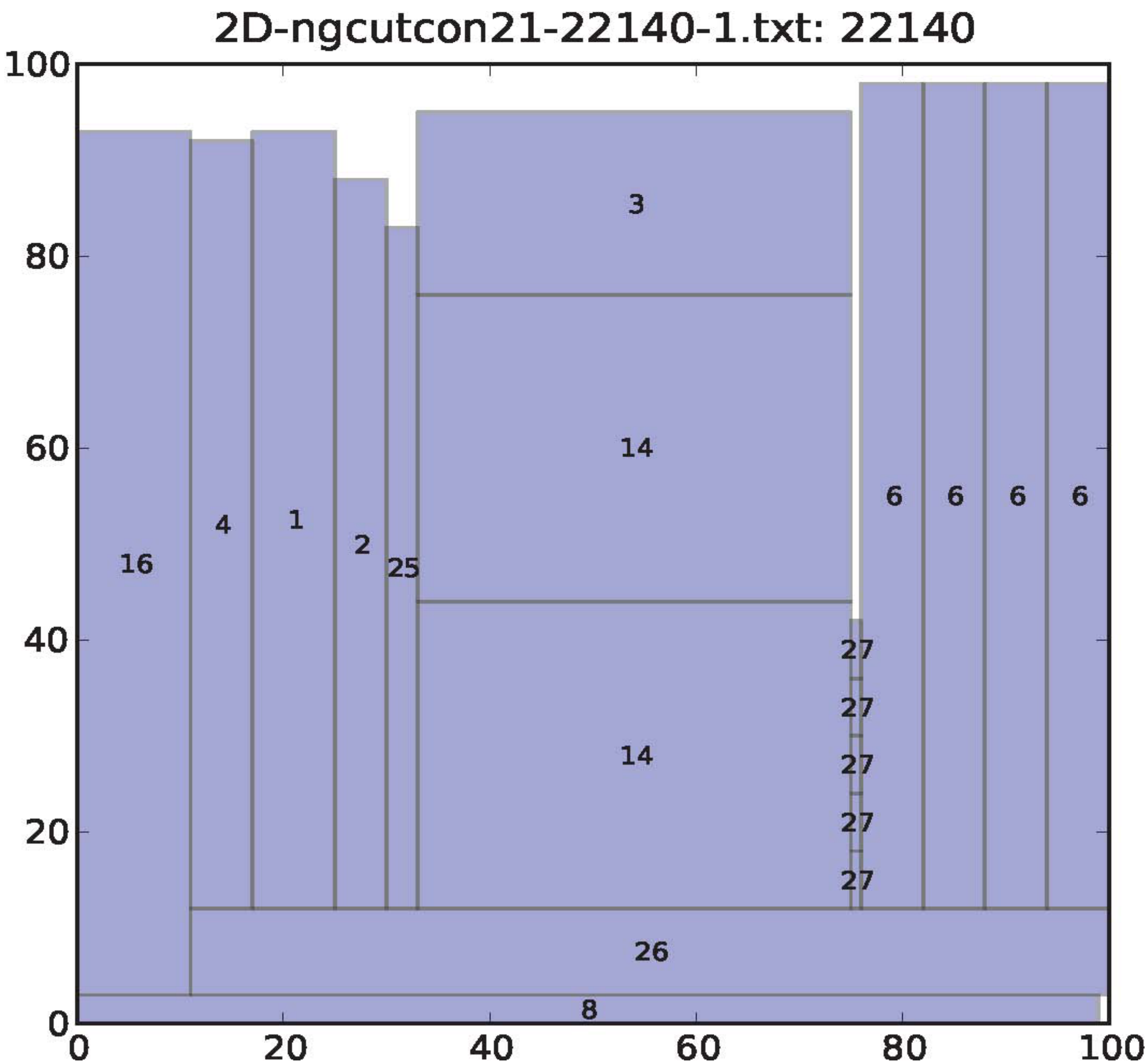
Minimum, average, and maximum solution times (secs) for BRKGA (BL-LB-L-4NR)

Problem	Min solution time (secs)	Avg solution time (secs)	Max solution time (secs)
From literature (optimal)	0.00	0.05	0.55
Large random	1.78	23.85	72.70
Zero-waste	0.01	82.21	808.03
Doubly constrained	0.00	1.16	16.87



New BKS for a 100 x100 doubly constrained instance of Fekete & Schepers (1997) of value **20678.** Previous best was **19657** by tabu search of Alvarez-Valdes et al., (2007).

30 types
30 rectangles



New BKS for a 100 x 100 doubly constrained instance Fekete & Schepers (1997) of value **22140**.

Previous BKS was **22011** by tabu search of Alvarez-Valdes et al. (2007).

29 types
97 rectangles

Some remarks

- We proposed a BRKGA heuristic for a constrained 2-dimensional orthogonal packing problem.
- Highlights:
  - Hybrid placement heuristics are coordinated by GA
  - Multiple populations evolve and exchange information
  - Modified fitness function
  - Parallel fitness evaluations
  - Some non-random starting solutions added to starting populations

Some remarks

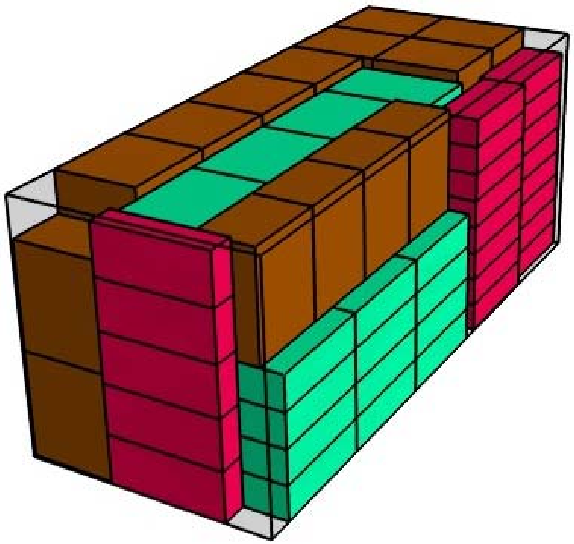
- Extensive computational experiments carried out.
- Highlights:
  - Layers improves only Bottom-Left
  - Left-Bottom improves Bottom-Left with layers
  - LB and BL with layers and 4 non-random starting solutions is best strategy
  - BRKGA finds better solutions than state of the art heuristics for a large number of instances
  - Several new best known solutions produced by the BRKGA

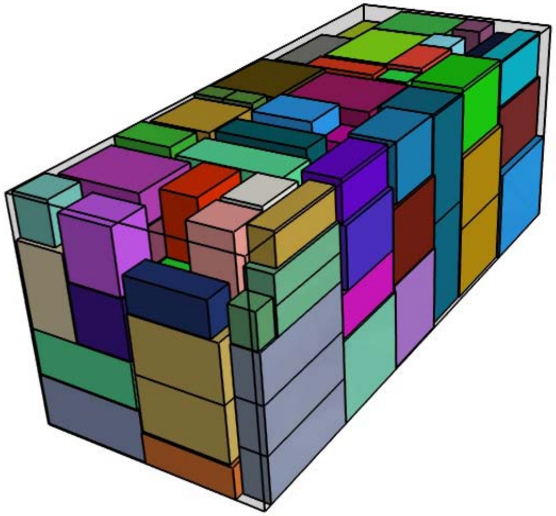
Some remarks

We have extended this to 3D packing:

J.F. Gonçalves and M.G.C.R., "A parallel multi-population biased random-key genetic algorithm for a container loading problem," Computers & Operations Research, vol. 29, pp. 179-190, 2012.

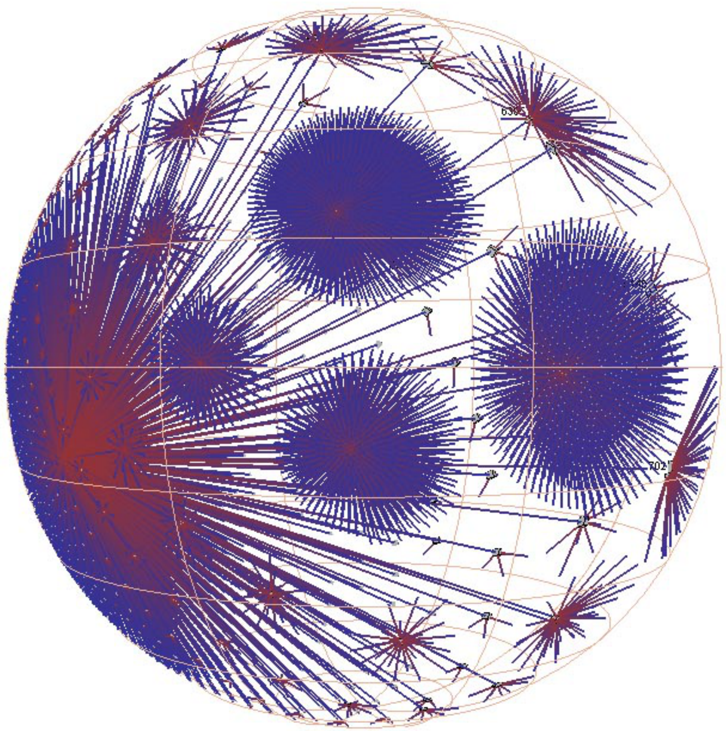
Tech report: http://www.research.att.com/~mgcr/doc/brkga-pack3d.pdf



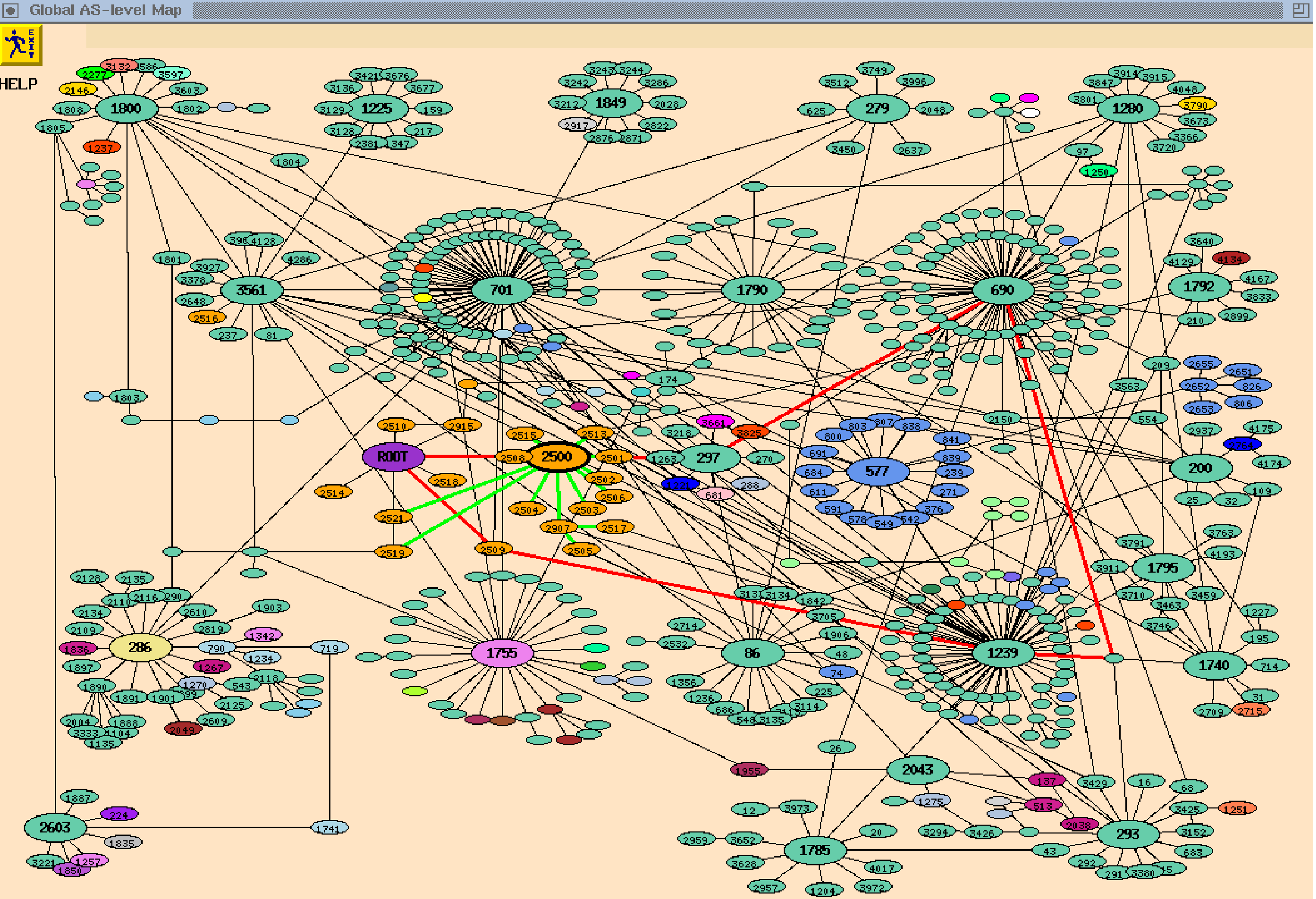


OSPF routing in IP networks

The Internet



- The Internet is composed of many (inter-connected) autonomous systems (AS).
- An AS is a network controlled by a single entity, e.g. ISP, university, corporation, country, ...


Global AS-level Map
EXIT
HELP
ROOT
2500
1800
1225
1849
279
1280
3561
701
1790
690
1792
297
577
200
1795
286
1755
86
1239
1740
2043
2603
1785
293

Routing

- A packet is sent from a origination router S to a destination router T.
- S and T may be in
  - same AS:
  - different ASes:

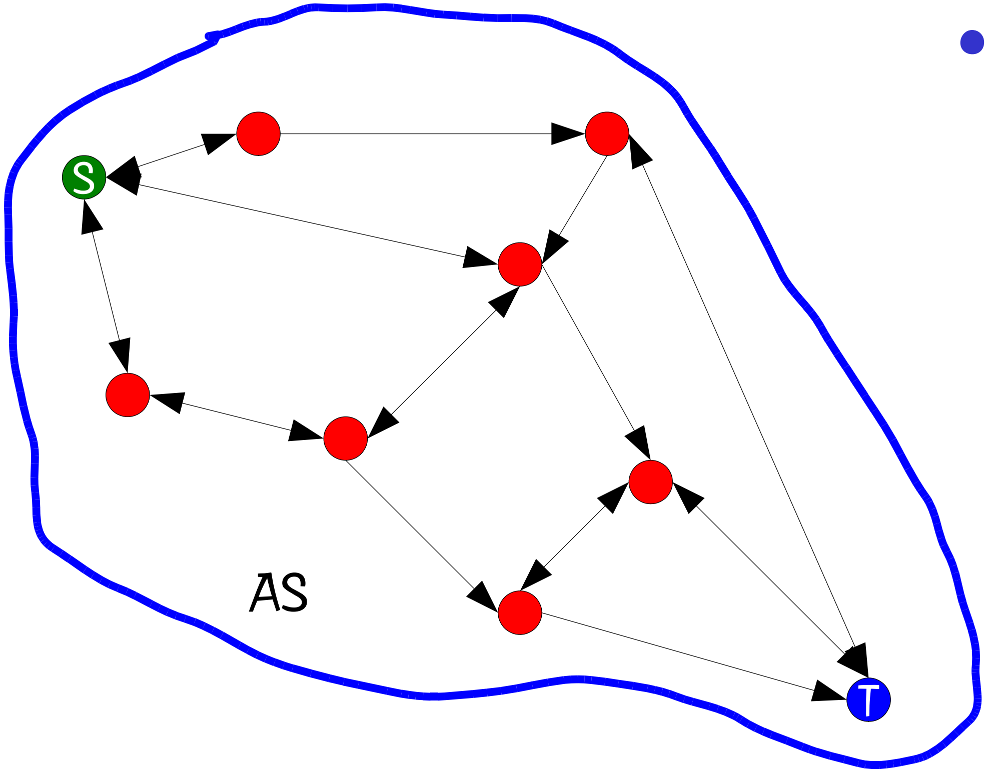
Routing

- A packet is sent from a origination router S to a destination router T.
- S and T may be in
  - same AS: IGP routing
  - different ASes:

Routing

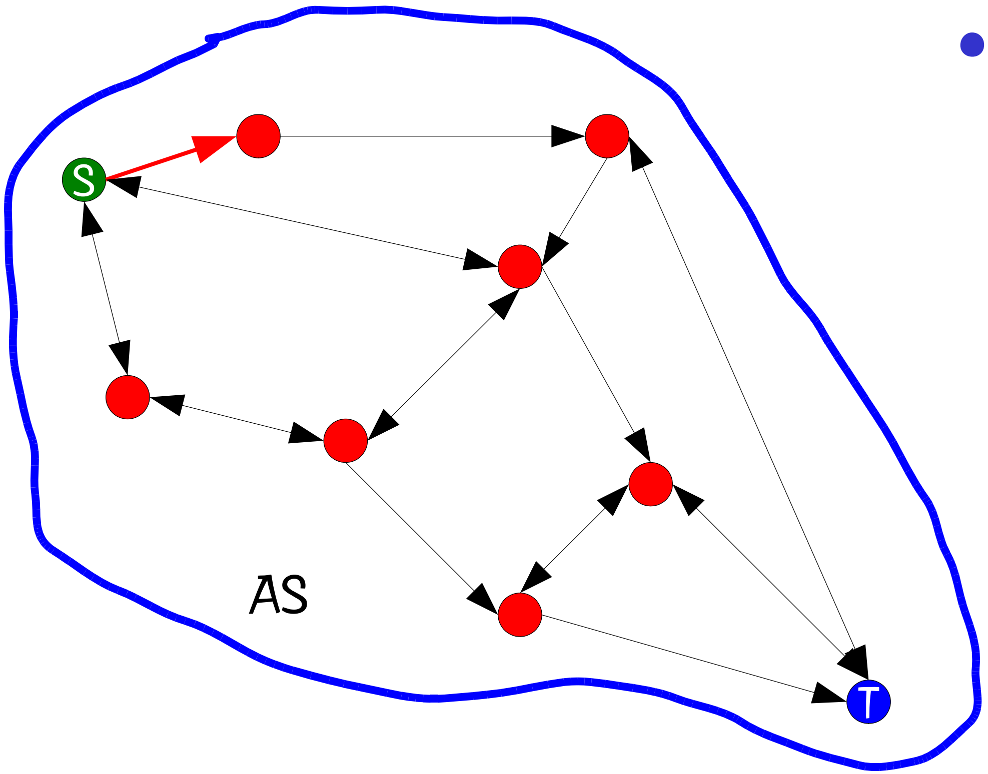
- A packet is sent from a origination router S to a destination router T.
- S and T may be in
  - same AS: IGP routing
  - different ASes: BGP routing

IGP Routing



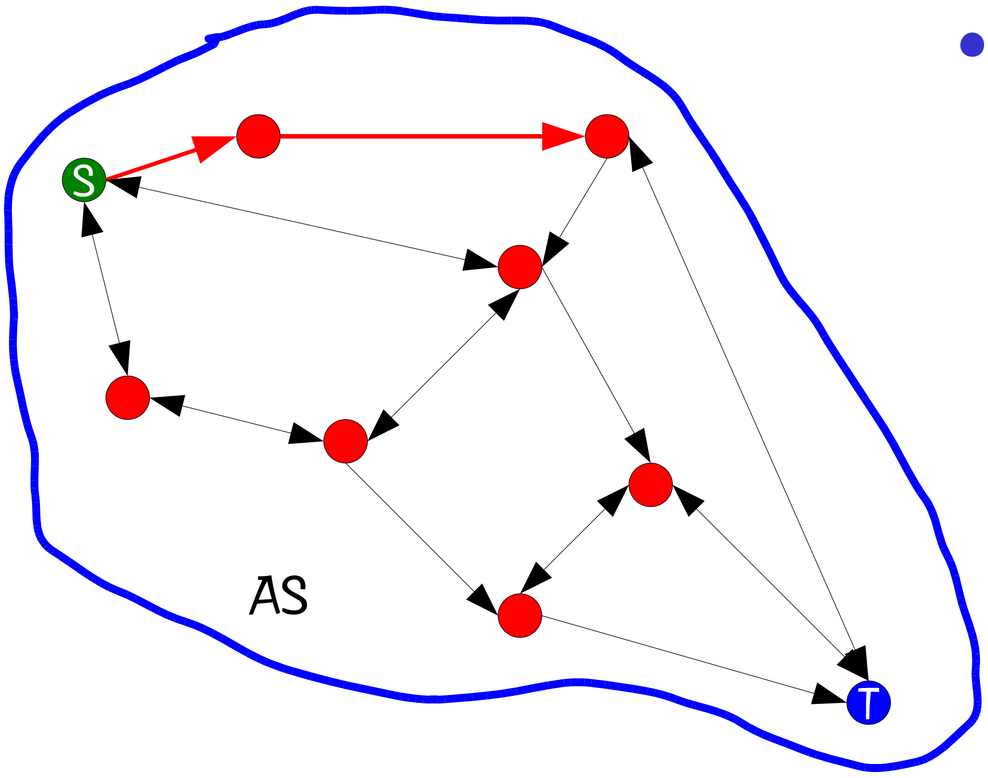
- IGP (interior gateway protocol) routing is concerned with routing within an AS.

IGP Routing



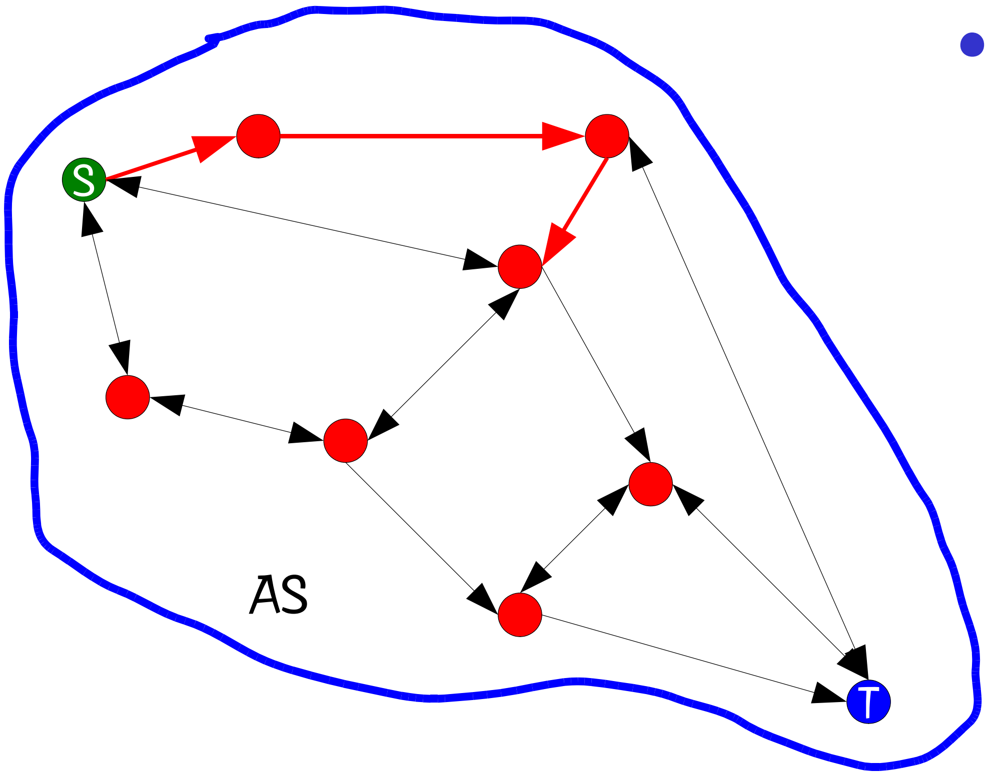
- IGP (interior gateway protocol) routing is concerned with routing within an AS.

IGP Routing



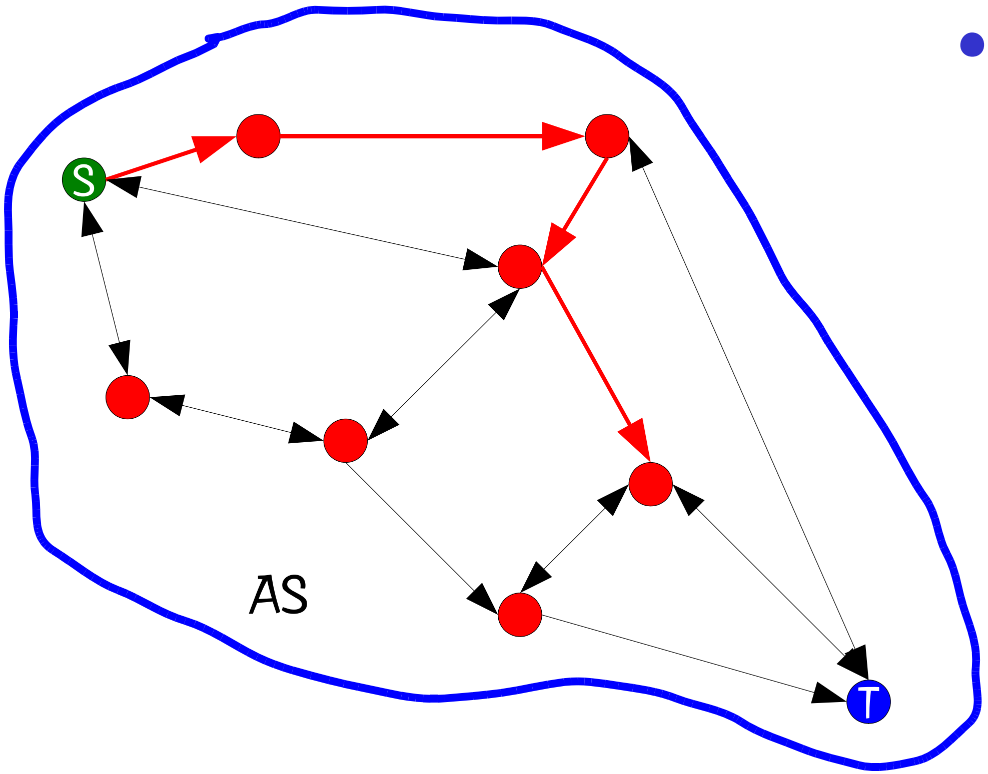
- IGP (interior gateway protocol) routing is concerned with routing within an AS.

IGP Routing



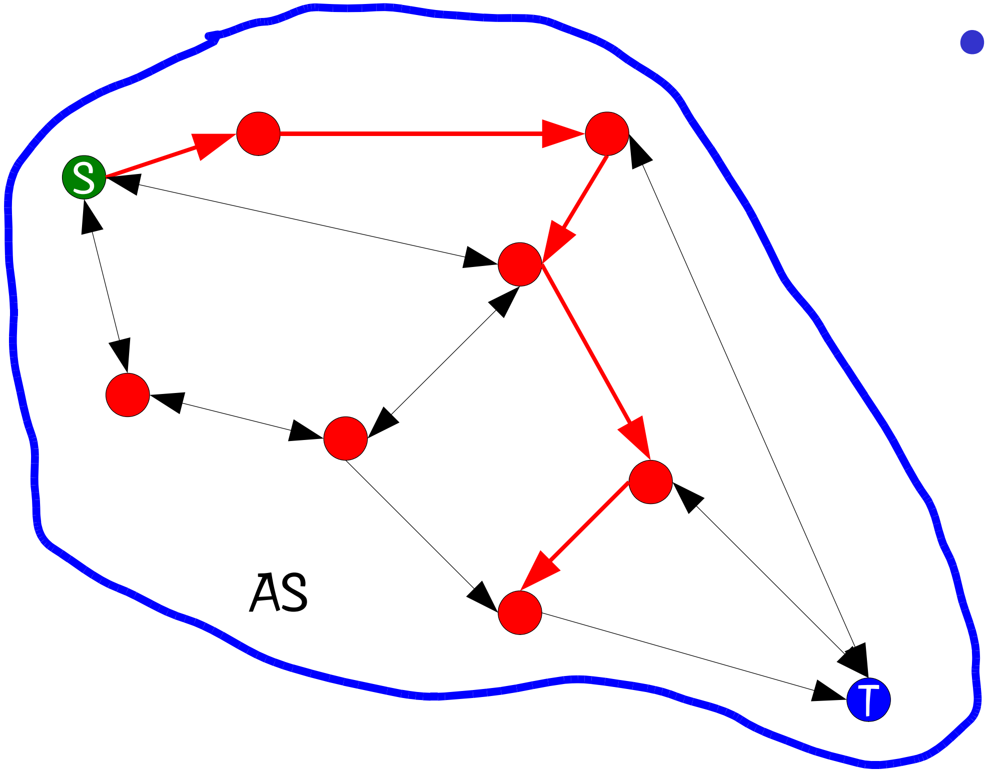
- IGP (interior gateway protocol) routing is concerned with routing within an AS.

IGP Routing



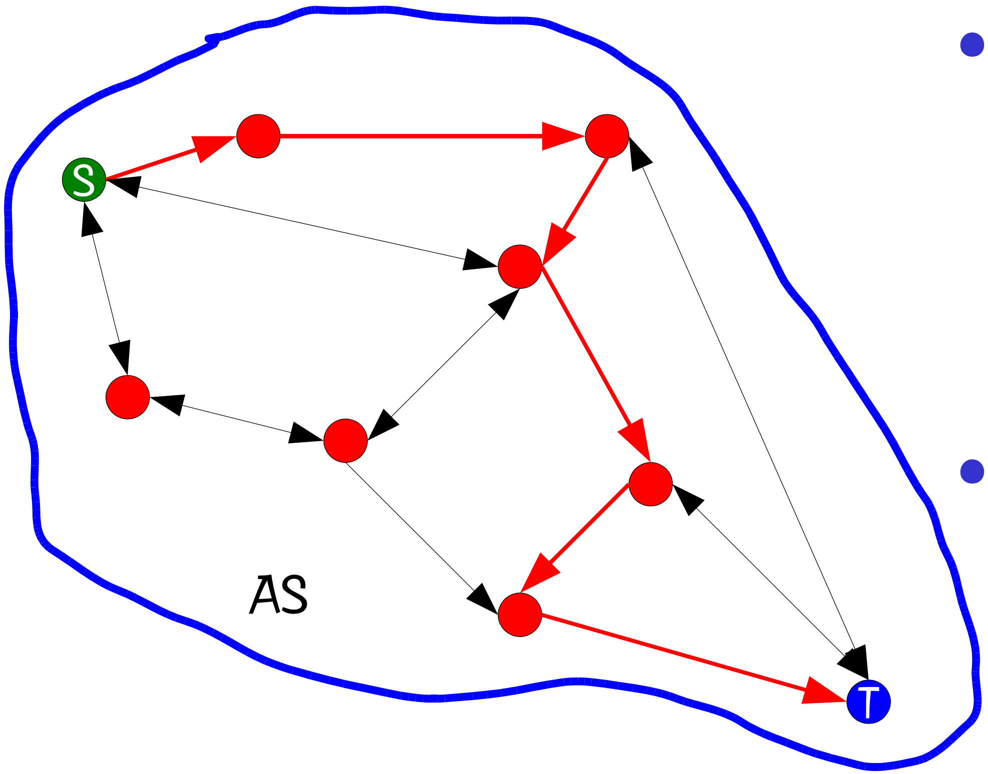
- IGP (interior gateway protocol) routing is concerned with routing within an AS.

IGP Routing



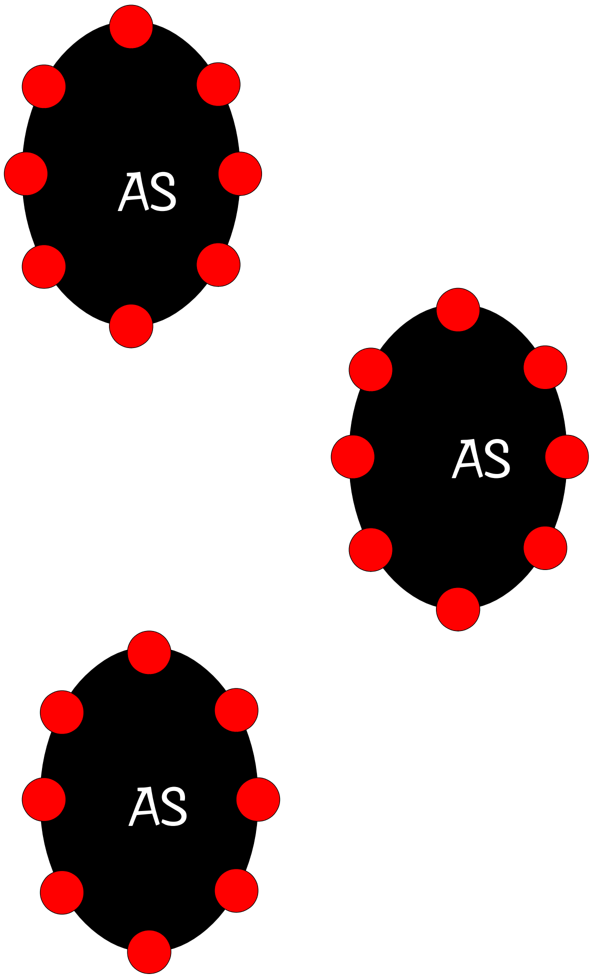
- IGP (interior gateway protocol) routing is concerned with routing within an AS.

IGP Routing



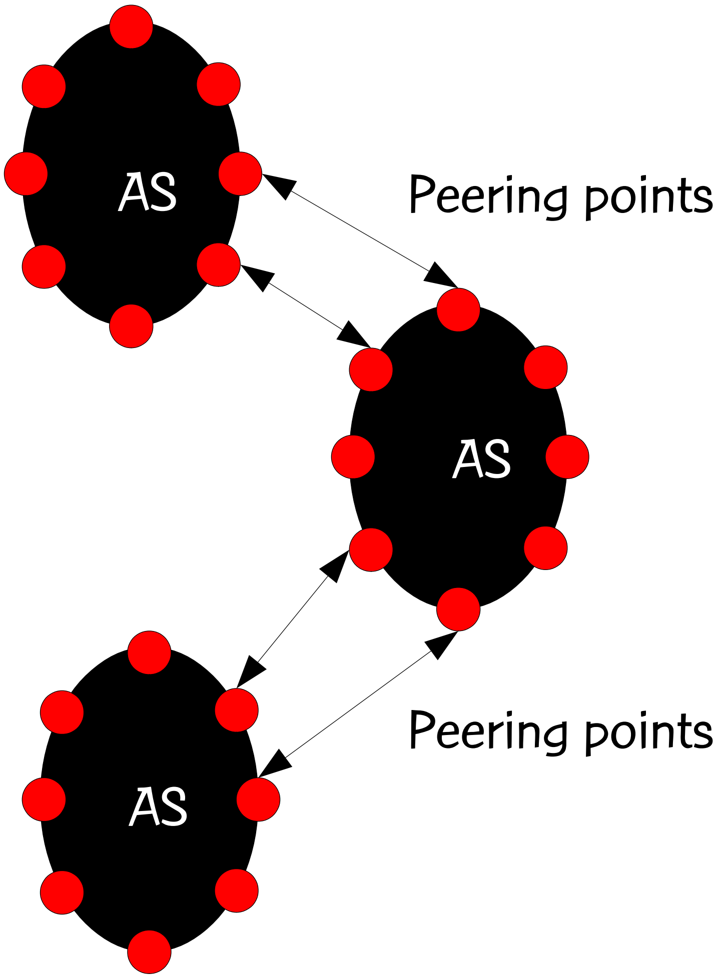
- IGP (interior gateway protocol) routing is concerned with routing within an AS.
- Routing decisions are made by AS operator.

BGP Routing



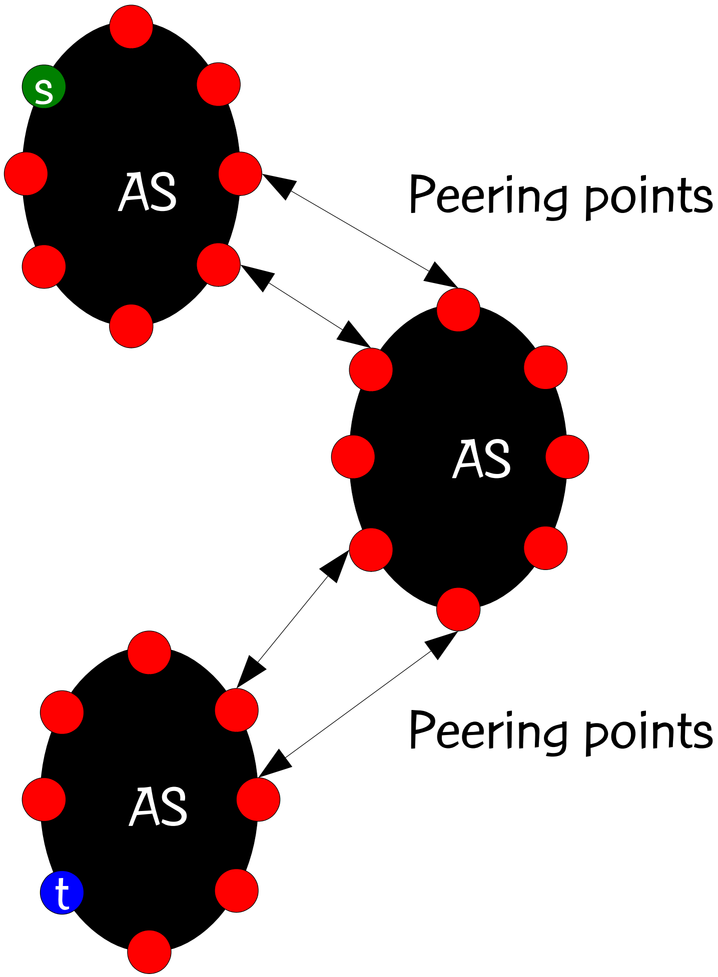
- BGP (border gateway protocol) routing deals with routing between different ASes.

BGP Routing



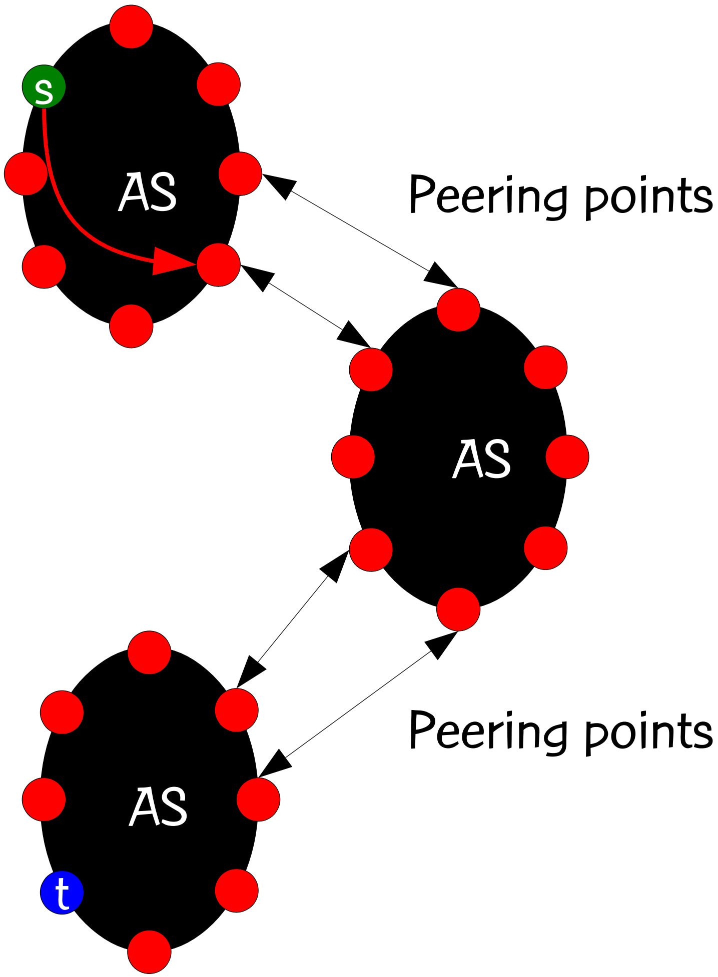
- BGP (border gateway protocol) routing deals with routing between different ASes.

BGP Routing



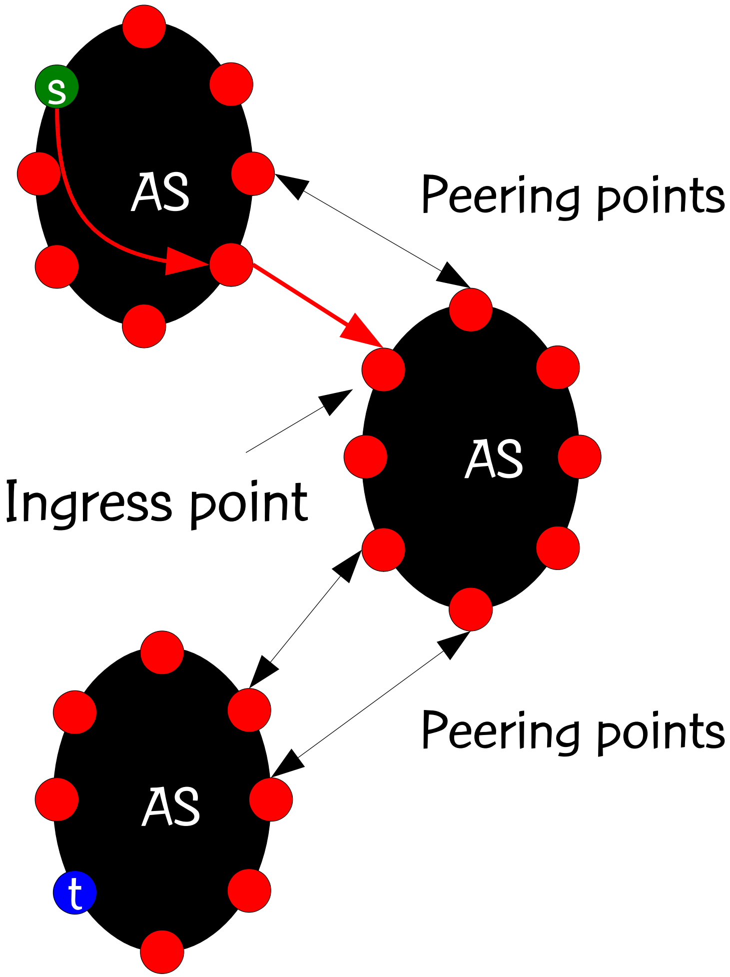
- BGP (border gateway protocol) routing deals with routing between different ASes.

BGP Routing



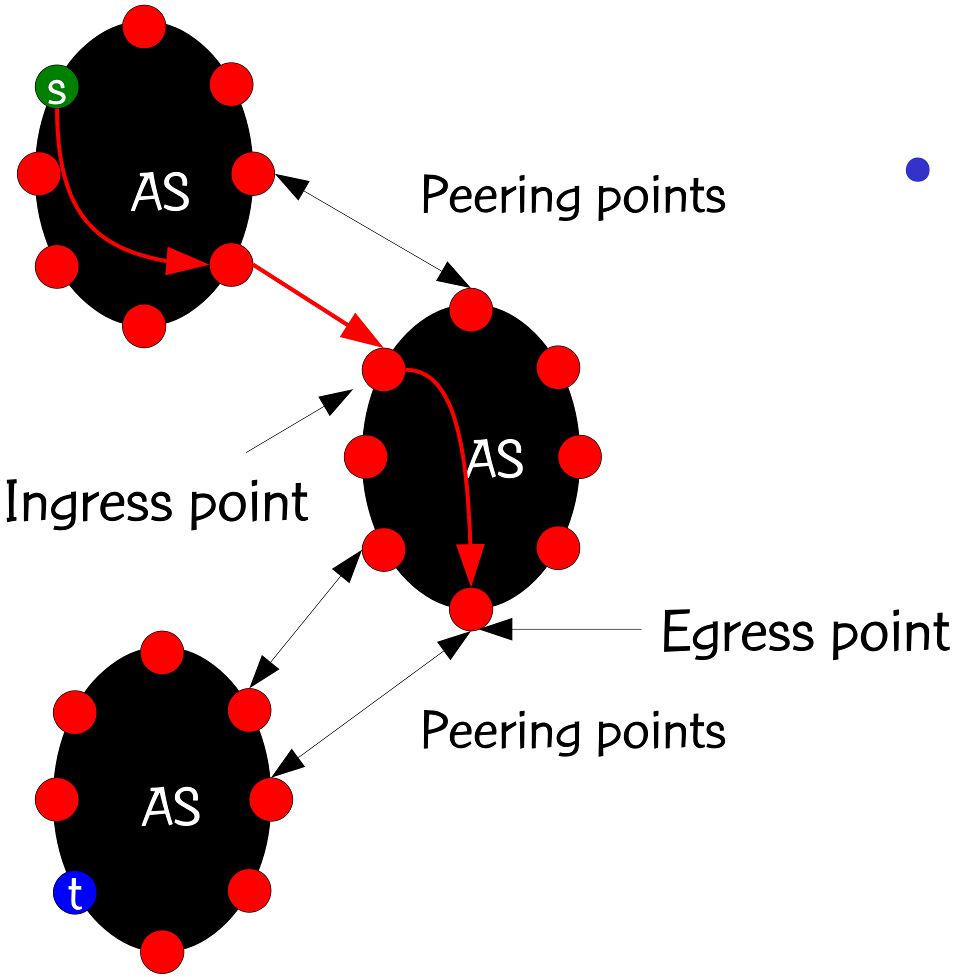
- BGP (border gateway protocol) routing deals with routing between different ASes.

BGP Routing



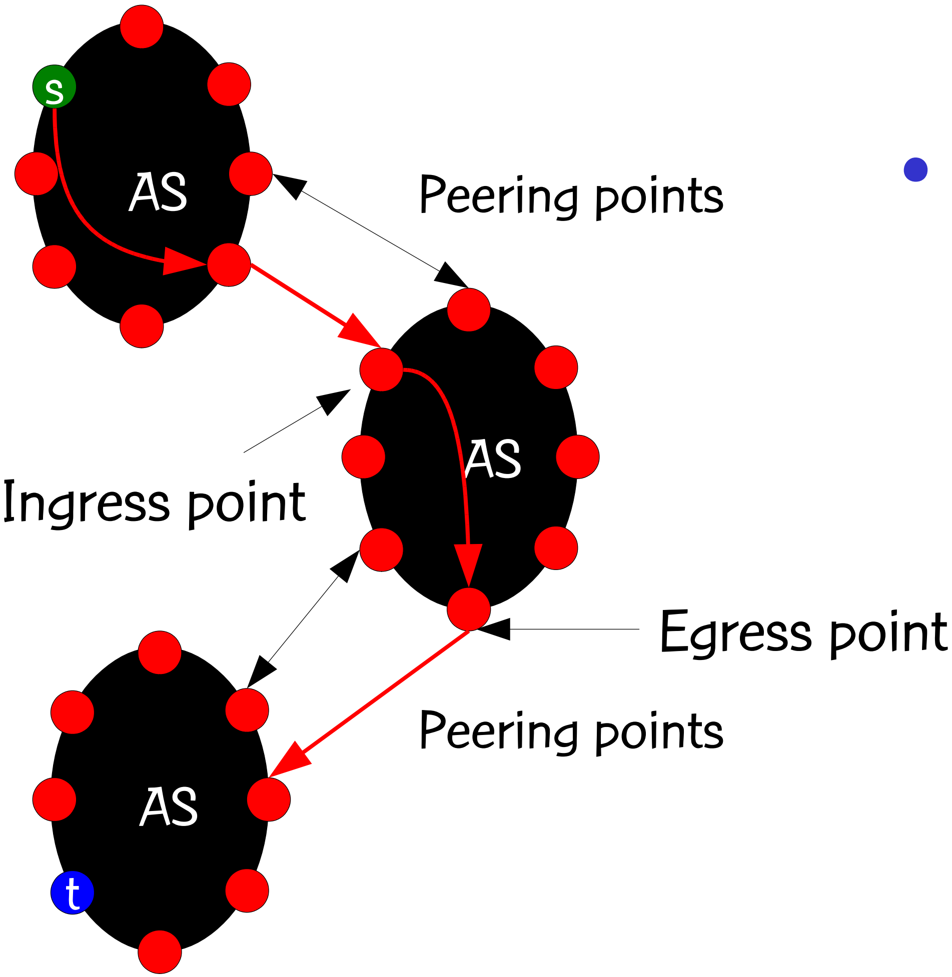
- BGP (border gateway protocol) routing deals with routing between different ASes.

BGP Routing



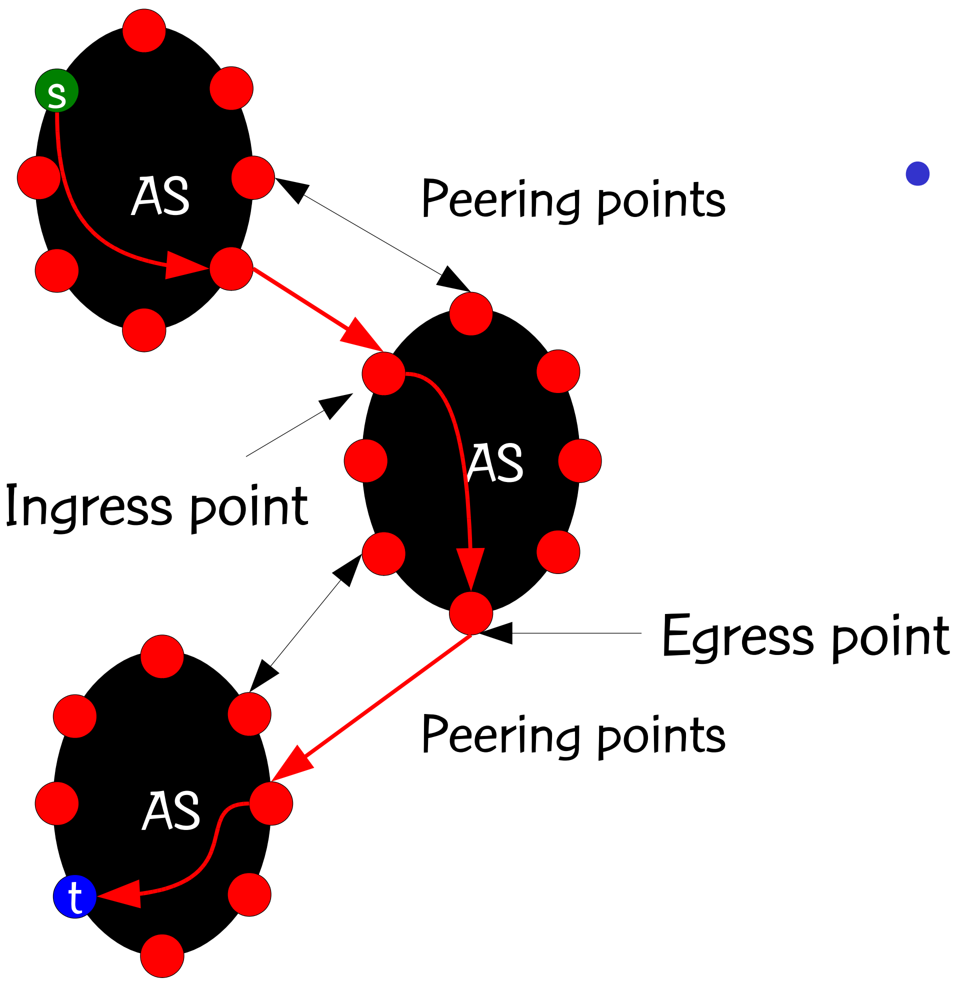
- BGP (border gateway protocol) routing deals with routing between different ASes.

BGP Routing



- BGP (border gateway protocol) routing deals with routing between different ASes.

BGP Routing



- BGP (border gateway protocol) routing deals with routing between different ASes.

IGP Routing

OSPF routing

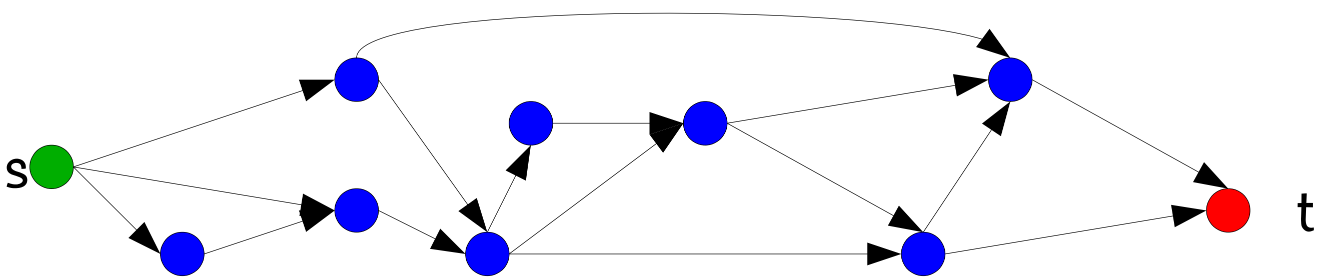
- Given a network $G = (N,A)$, where N is the set of routers and A is the set of links.

OSPF routing

- Given a network $G = (N,A)$, where N is the set of routers and A is the set of links.
- The OSPF (open shortest path first) routing protocol assumes each link a has a weight w(a) assigned to it so that a packet from a source router s to a destination router t is routed on a shortest weight path from s to t.

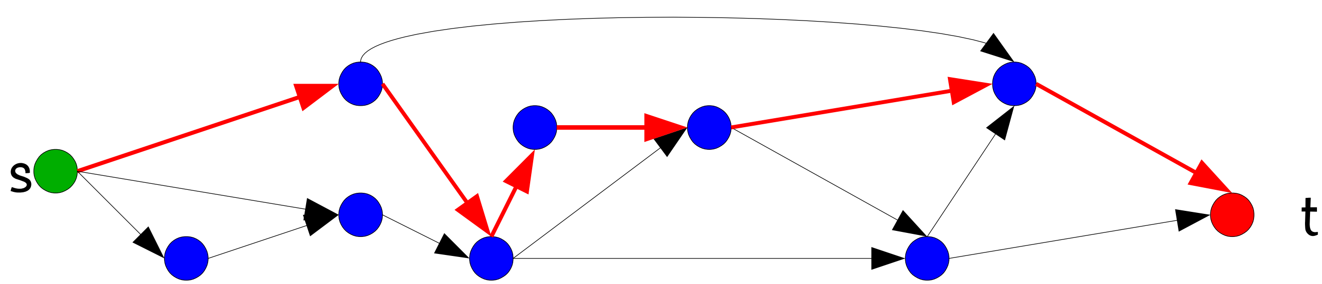
OSPF routing

- Given a network $G = (N,A)$, where N is the set of routers and A is the set of links.
- The OSPF (open shortest path first) routing protocol assumes each link a has a weight w(a) assigned to it so that a packet from a source router s to a destination router t is routed on a shortest weight path from s to t.



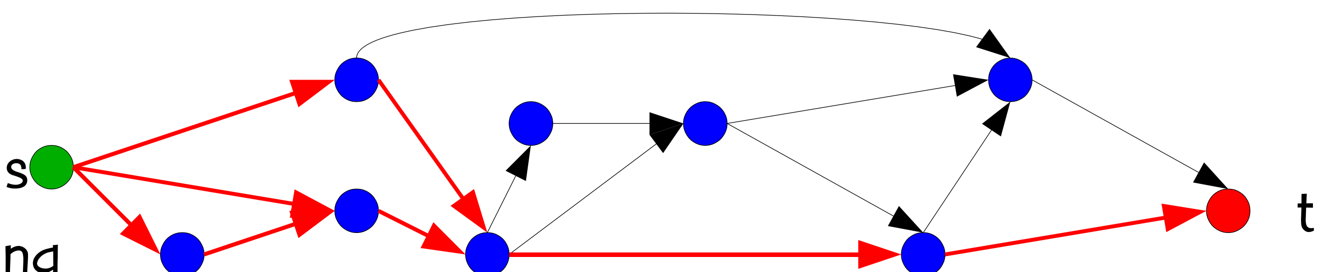
OSPF routing

- Given a network $G = (N,A)$, where N is the set of routers and A is the set of links.
- The OSPF (open shortest path first) routing protocol assumes each link a has a weight w(a) assigned to it so that a packet from a source router s to a destination router t is routed on a shortest weight path from s to t.



OSPF routing

- Given a network $G = (N,A)$, where N is the set of routers and A is the set of links.
- The OSPF (open shortest path first) routing protocol assumes each link a has a weight w(a) assigned to it so that a packet from a source router s to a destination router t is routed on a shortest weight path from s to t.



Traffic splitting

OSPF routing

- By setting OSPF weights appropriately, one can do traffic engineering, i.e. route traffic so as to optimize some objective (e.g. minimize congestion, maximize throughput, etc.).
- Some recent papers on this topic:
  - Fortz & Thorup (2000, 2004)
  - Ramakrishnan & Rodrigues (2001)
  - Sridharan, Guérin, & Diot (2002)
  - Fortz, Rexford, & Thorup (2002)
  - Ericsson, Resende, & Pardalos (2002)
  - Buriol, Resende, Ribeiro, & Thorup (2002, 2005)
  - Reis, Ritt, Buriol, & Resende (2011)

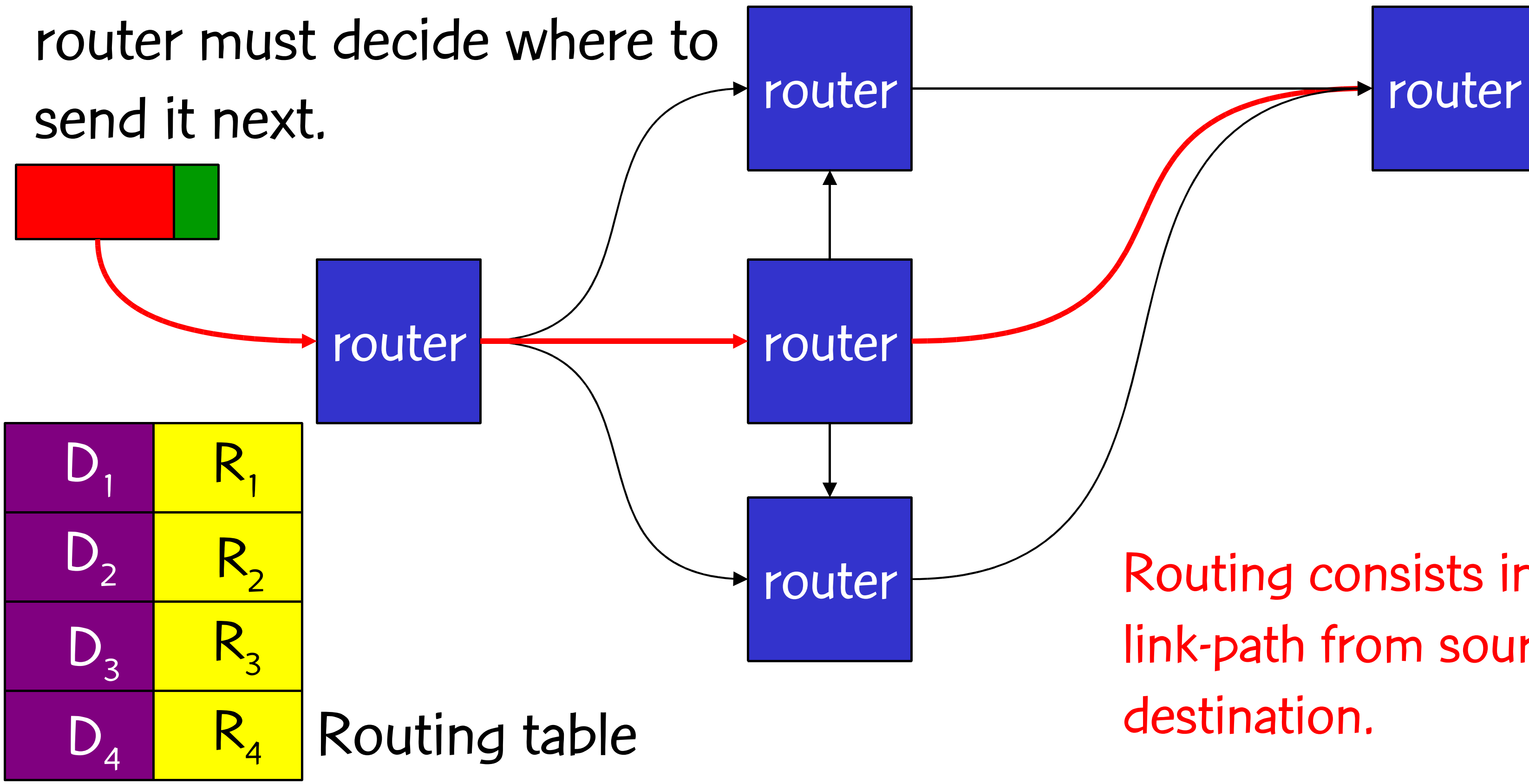
OSPF routing

- By setting OSPF weights appropriately, one can do traffic engineering, i.e. route traffic so as to optimize some objective (e.g. minimize congestion, maximize throughput, etc.).
- Some recent papers on this topic:
  - Fortz & Thorup (2000, 2004)
  - Ramakrishnan & Rodrigues (2001)
  - Sridharan, Guérin, & Diot (2002)
  - Fortz, Rexford, & Thorup (2002)
  - Ericsson, Resende, & Pardalos (2002)
  - Buriol, Resende, Ribeiro, & Thorup (2002, 2005)
  - Reis, Ritt, Buriol & Resende (2011)

Packet routing

When packet arrives at router, router must decide where to send it next.

Packet's final destination.



Routing consists in finding a link-path from source to destination.

OSPF routing

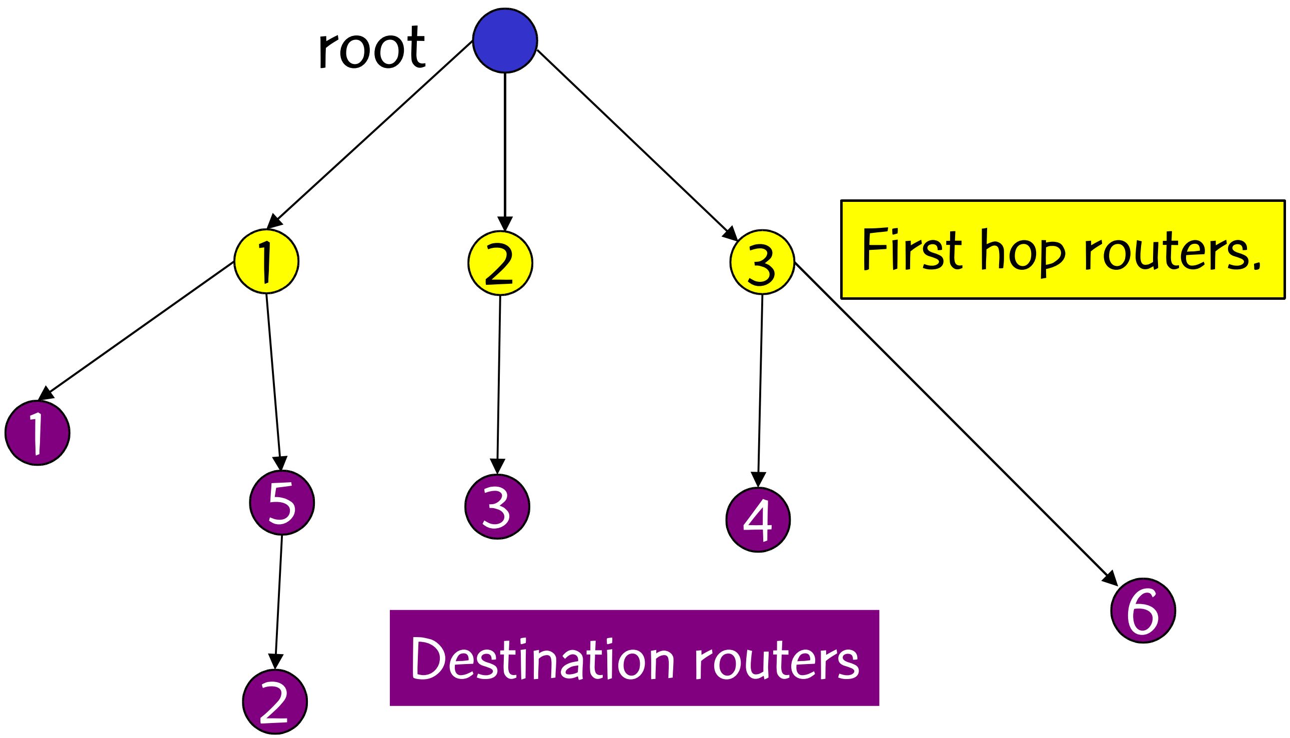
- Assign an integer weight $\in [1, w_{max}]$ to each link in AS. In general, $w_{max} = 65535 = 2^{16} - 1$.
- Each router computes tree of shortest weight paths to all other routers in the AS, with itself as the root, using Dijkstra's algorithm.

OSPF routing

Routing table is filled
with first hop routers
for each possible destination.

Routing table

D_1	R_1
D_2	R_1
D_3	R_2
D_4	R_3
D_5	R_1
D_6	R_3

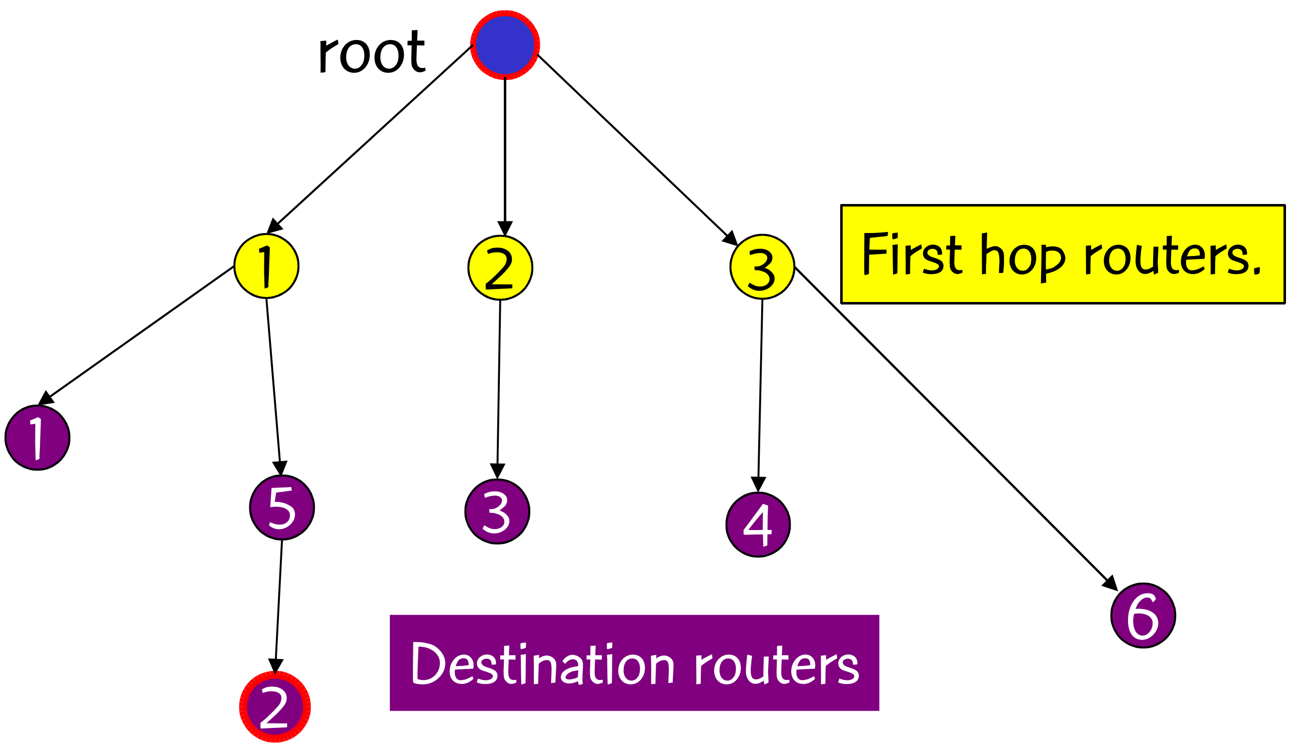


OSPF routing

Routing table is filled
with first hop routers
for each possible destination.

Routing table

D_1	R_1
D_2	R_1
D_3	R_2
D_4	R_3
D_5	R_1
D_6	R_3

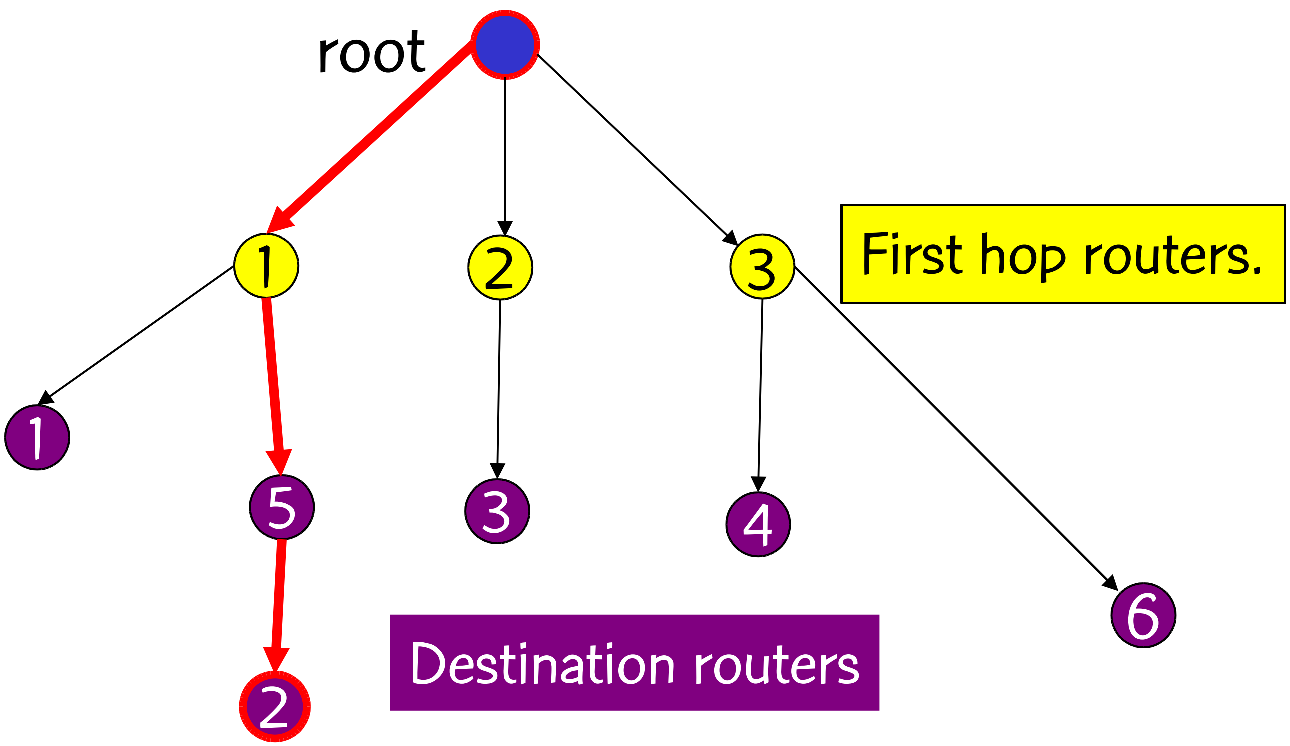


OSPF routing

Routing table is filled
with first hop routers
for each possible destination.

Routing table

D_1	R_1
D_2	R_1
D_3	R_2
D_4	R_3
D_5	R_1
D_6	R_3

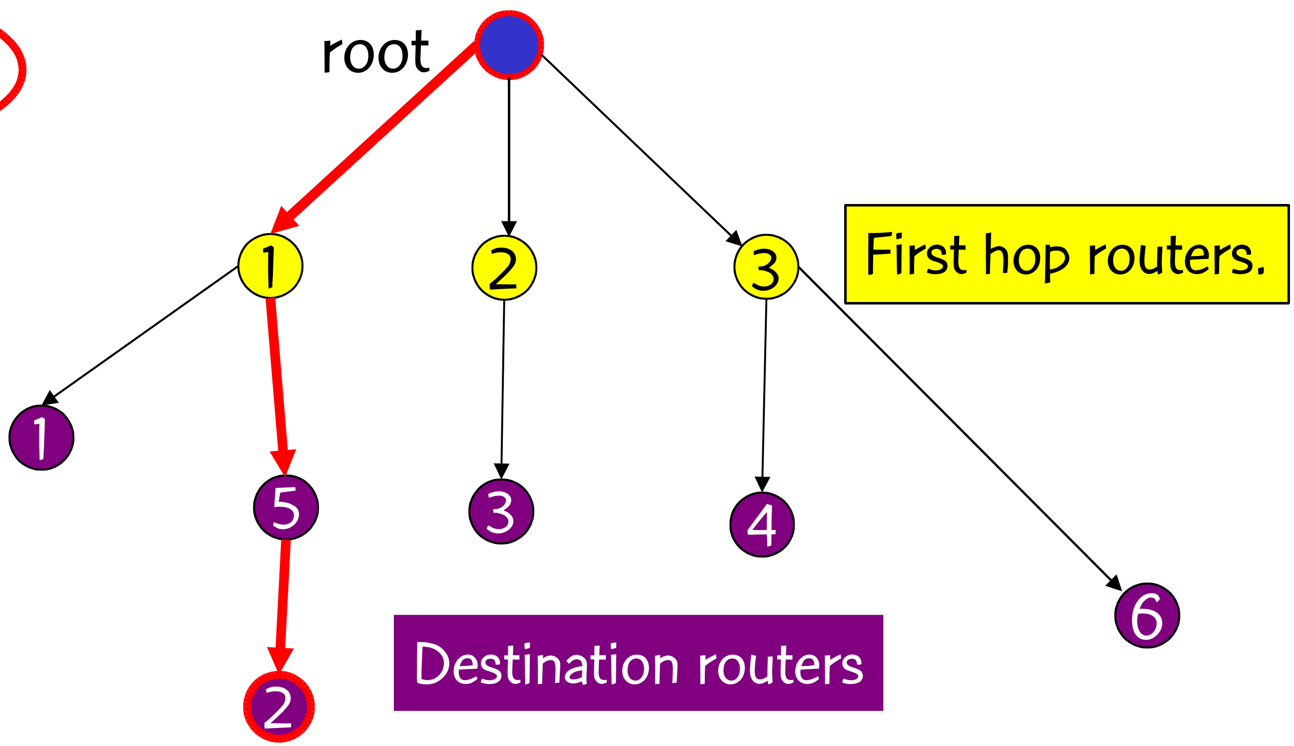


OSPF routing

Routing table is filled
with first hop routers
for each possible destination.

Routing table

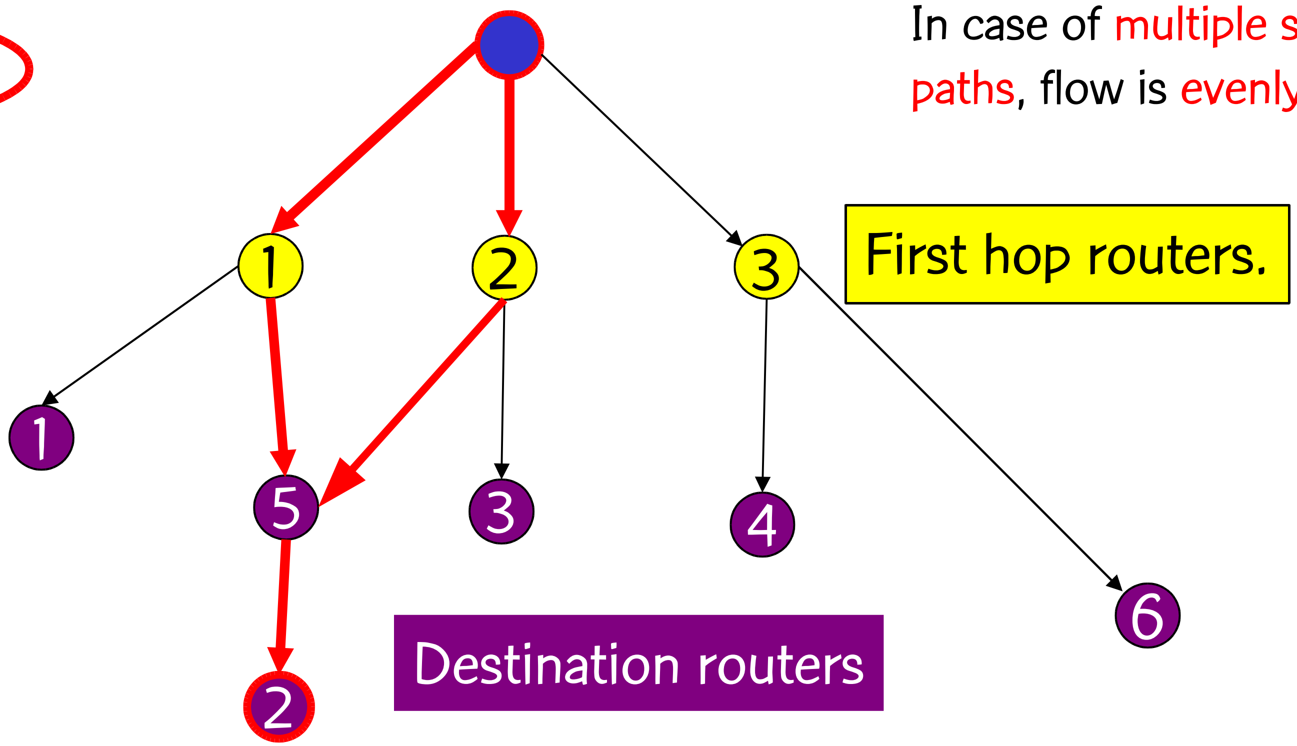
D_1	R_1
D_2	R_1
D_3	R_2
D_4	R_3
D_5	R_1
D_6	R_3



OSPF routing

Routing table

D_1	R_1
D_2	R_1, R_2
D_3	R_2
D_4	R_3
D_5	R_1
D_6	R_3



Routing table is filled with first hop routers for each possible destination. In case of multiple shortest paths, flow is evenly split.

OSPF weight setting

- OSPF weights are assigned by network operator.
  - CISCO assigns, by default, a weight proportional to the inverse of the link bandwidth (Inv Cap).
  - If all weights are unit, the weight of a path is the number of hops in the path.
- We propose two BRKGA to find good OSPF weights.

Minimization of congestion

- Consider the directed capacitated network $G = (N,A,c)$, where N are routers, A are links, and c_a is the capacity of link $a \in A$.
- We use the measure of Fortz & Thorup (2000) to compute congestion:

$$\Phi = \Phi_1(l_1) + \Phi_2(l_2) + \ldots + \Phi_{|A|}(l_{|A|})$$

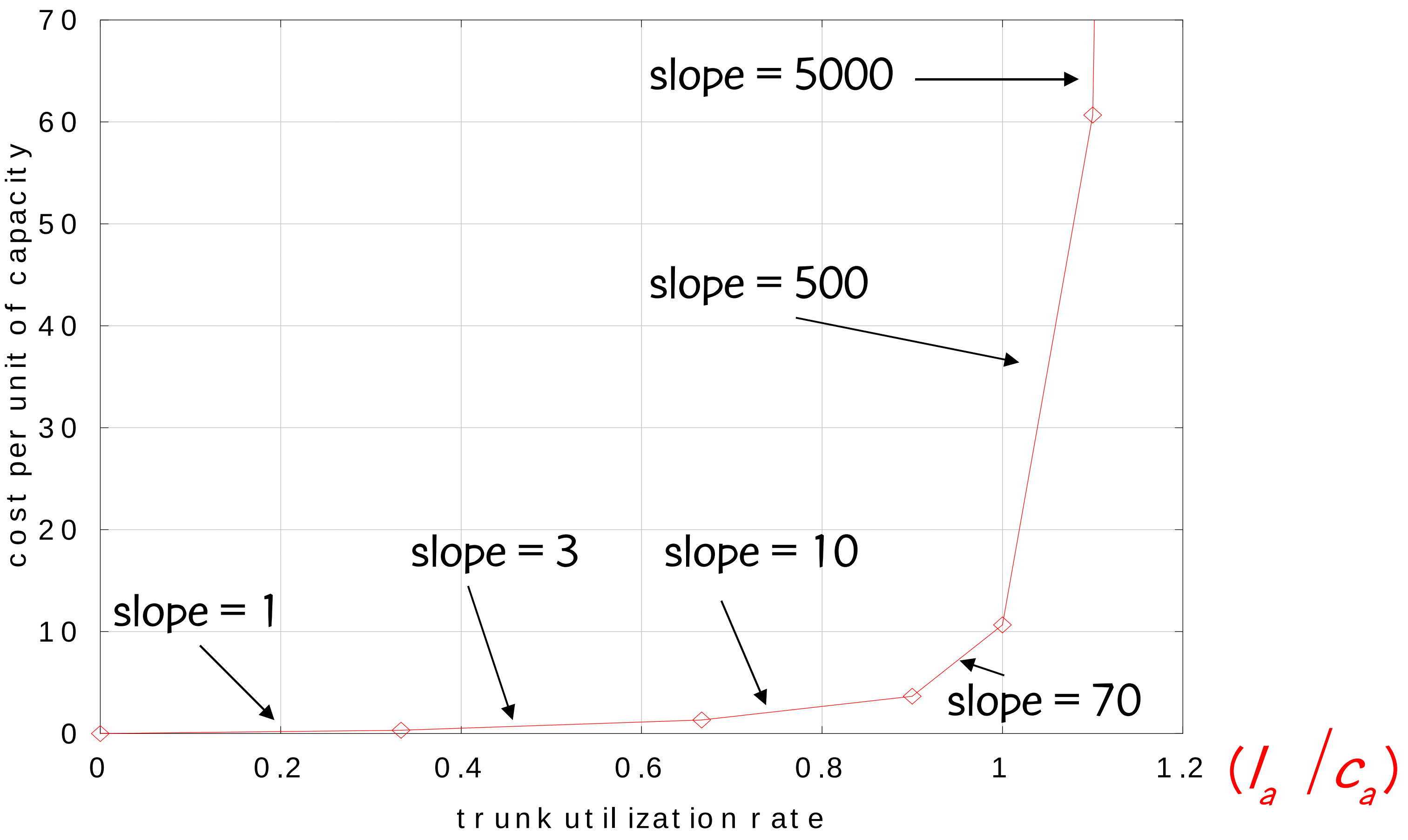
where l_a is the load on link $a \in A$,

$\Phi_a(l_a)$ is piecewise linear and convex,

$\Phi_a(0) = 0$, for all $a \in A$.

at&t
Your world. Delivered.

Piecewise linear and convex $\Phi_a(l_a)$ link congestion measure



OSPF weight setting problem

- Given a directed network $G = (N, A)$ with link capacities $c_a \in A$ and demand matrix $D = (d_{s,t})$ specifying a demand to be sent from node s to node t:
  - Assign weights $w_a \in [1, w_{max}]$ to each link $a \in A$, such that the objective function Φ is minimized when demand is routed according to the OSPF protocol.

BRKGA for OSPF routing in IP networks

M. Ericsson, M.G.C.R., & P.M. Pardalos, "A genetic algorithm for the weight setting problem in OSPF routing," J. of Combinatorial Optimization, vol. 6, pp. 299–333, 2002.

Tech report version:

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

BRKGA for OSPF routing in IP networks

Ericsson, R., & Pardalos (J. Comb. Opt., 2002)

- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.

BRKGA for OSPF routing in IP networks

Ericsson, R., & Pardalos (J. Comb. Opt., 2002)

- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.
- Decoding:

BRKGA for OSPF routing in IP networks

Ericsson, R., & Pardalos (J. Comb. Opt., 2002)

- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.
- Decoding:
  - For $i = 1, \ldots, N$: set $w(i) = \text{ceil}(X(i) \times w_{max})$

BRKGA for OSPF routing in IP networks

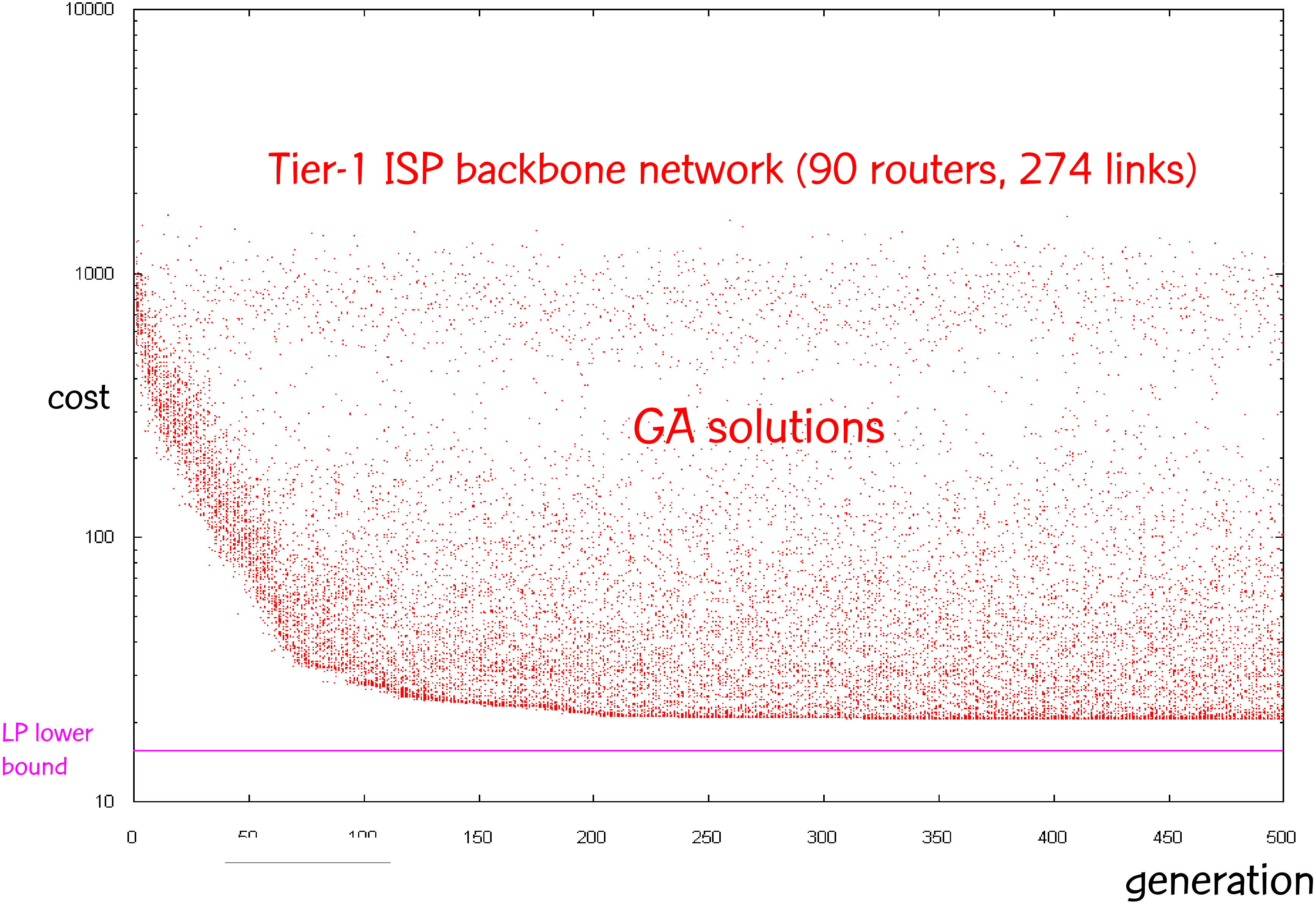
Ericsson, R., & Pardalos (J. Comb. Opt., 2002)

- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.
- Decoding:
  - For $i = 1, \ldots, N$: set $w(i) = \text{ceil}\,(X(i) \times w_{max})$
  - Compute shortest paths and route traffic according to OSPF.

BRKGA for OSPF routing in IP networks

Ericsson, R., & Pardalos (J. Comb. Opt., 2002)

- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.
- Decoding:
  - For i = 1, ..., N: set $w(i) = \text{ceil}(X(i) \times w_{max})$
  - Compute shortest paths and route traffic according to OSPF.
  - Compute load on each link, compute link congestion, add up all link congestions to compute network congestion.

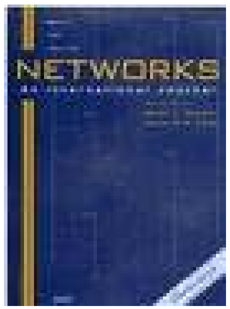


10000
1000
100
10
cost
Tier-1 ISP backbone network (90 routers, 274 links)
GA solutions
LP lower
bound
0
150
200
250
300
350
400
450
500
generation

Tier-1 ISP backbone network (90 routers, 274 links)

Weight setting with GA permits a 50% increase in traffic volume w.r.t. weight setting with the Inverse Capacity rule.

Improved BRKGA for OSPF routing in IP networks



L.S. Buriol, M.G.C.R., C.C. Ribeiro, and M. Thorup, "A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing," Networks, vol. 46, pp. 36–56, 2005.

Tech report version:

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

at&t
Your world. Delivered.

Improved BRKGA for OSPF routing in IP networks

Buriol, R., Ribeiro, and Thorup (Networks, 2005)

- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.

Improved BRKGA for OSPF routing in IP networks

Buriol, R., Ribeiro, and Thorup (Networks, 2005)

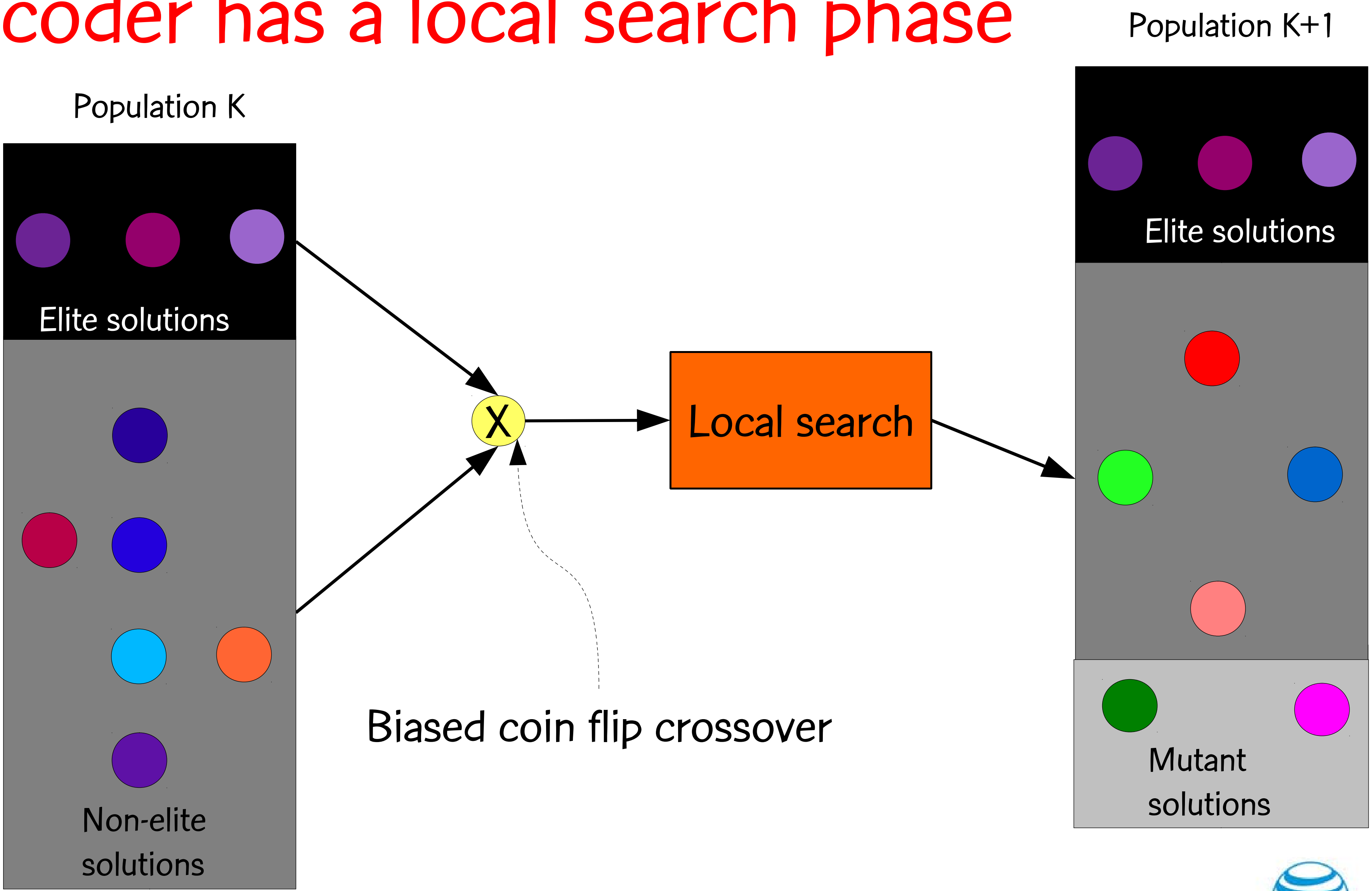
- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.
- Decoder:
  - For $i = 1, \ldots, N$: set $w(i) = \text{ceil}\,(\, X(i) \times w_{max}\,)$
  - Compute shortest paths and route traffic according to OSPF.
  - Compute load on each link, compute link congestion, add up all link congestions to compute network congestion.

Improved BRKGA for OSPF routing in IP networks

Buriol, R., Ribeiro, and Thorup (Networks, 2005)

- Encoding:
  - A vector X of N random keys, where N is the number of links. The i-th random key corresponds to the i-th link weight.
- Decoder:
  - For $i = 1, \ldots, N$: set $w(i) = \text{ceil}\,(\,X(i) \times w_{max}\,)$
  - Compute shortest paths and route traffic according to OSPF.
  - Compute load on each link, compute link congestion, add up all link congestions to compute network congestion.
  - Apply fast local search to improve weights.

Decoder has a local search phase



Fast local search

- Let A^* be the set of five arcs $a \in A$ having largest Φ_a values.

Fast local search

- Let A^* be the set of five arcs $a \in A$ having largest Φ_a values.
- Scan arcs $a \in A^*$ from largest to smallest Φ_a:

Fast local search

- Let A^* be the set of five arcs $a \in A$ having largest Φ_a values.
- Scan arcs $a \in A^*$ from largest to smallest Φ_a:
  - Increase arc weight, one unit at a time, in the range

$$[w_a, w_a + \lceil (w_{max} - w_a)/4 \rceil]$$

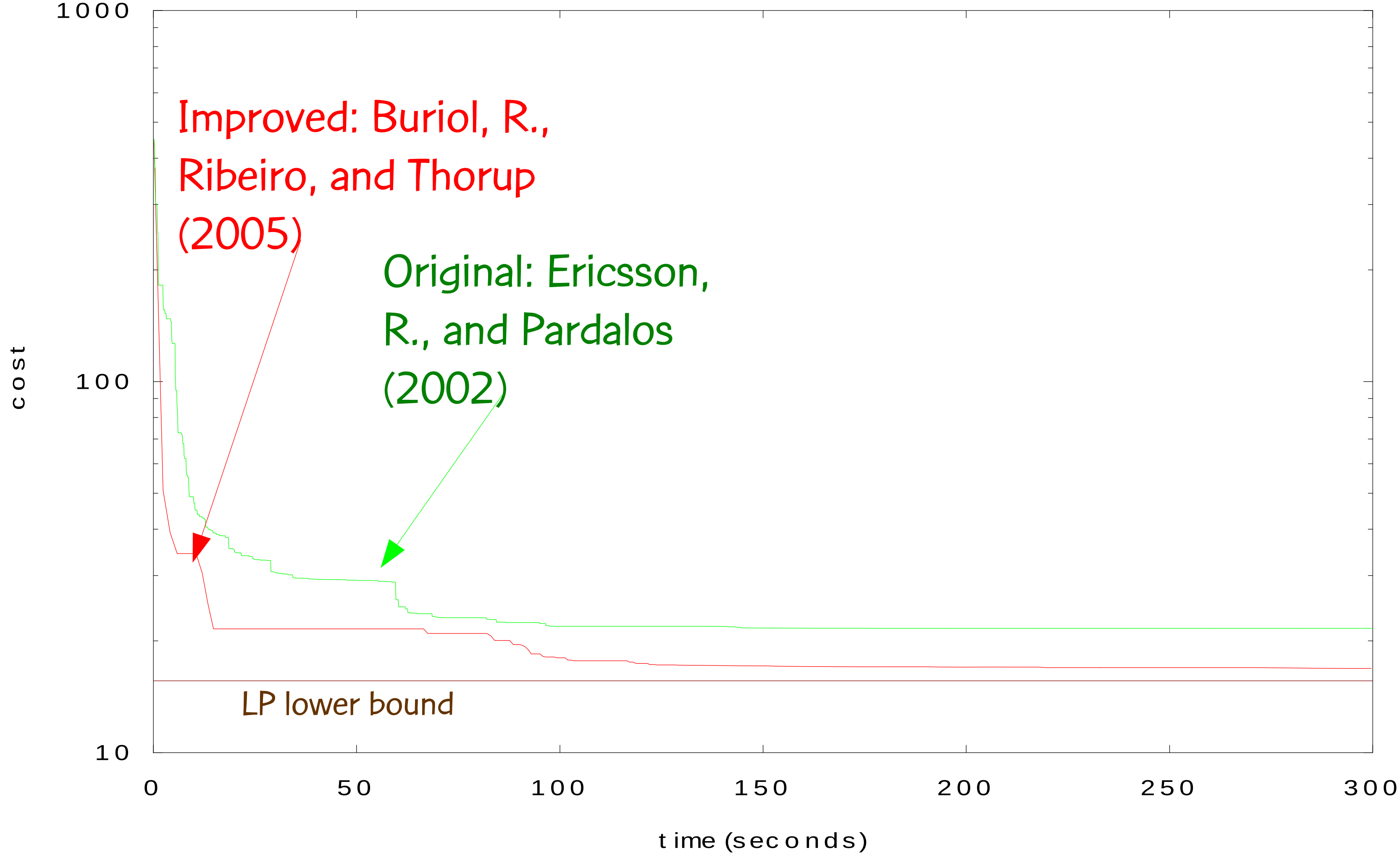
Fast local search

- Let A^* be the set of five arcs $a \in A$ having largest Φ_a values.
- Scan arcs $a \in A^*$ from largest to smallest Φ_a:
  - Increase arc weight, one unit at a time, in the range

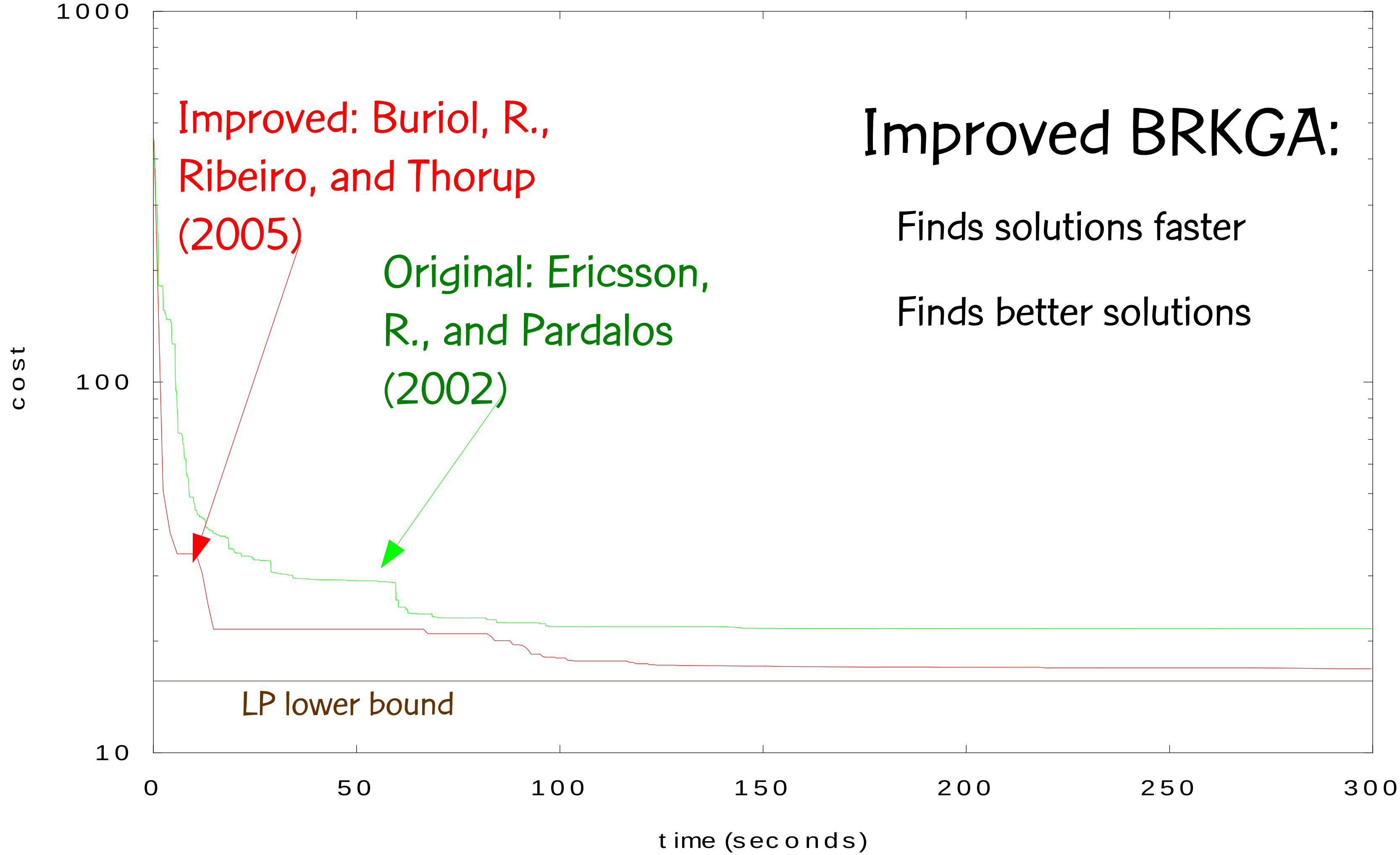
    $$[w_a , w_a + \lceil (w_{max} - w_a)/4 \rceil]$$

  - If total cost Φ is reduced, restart local search.

Effect of decoder with fast local search



Effect of decoder with fast local search

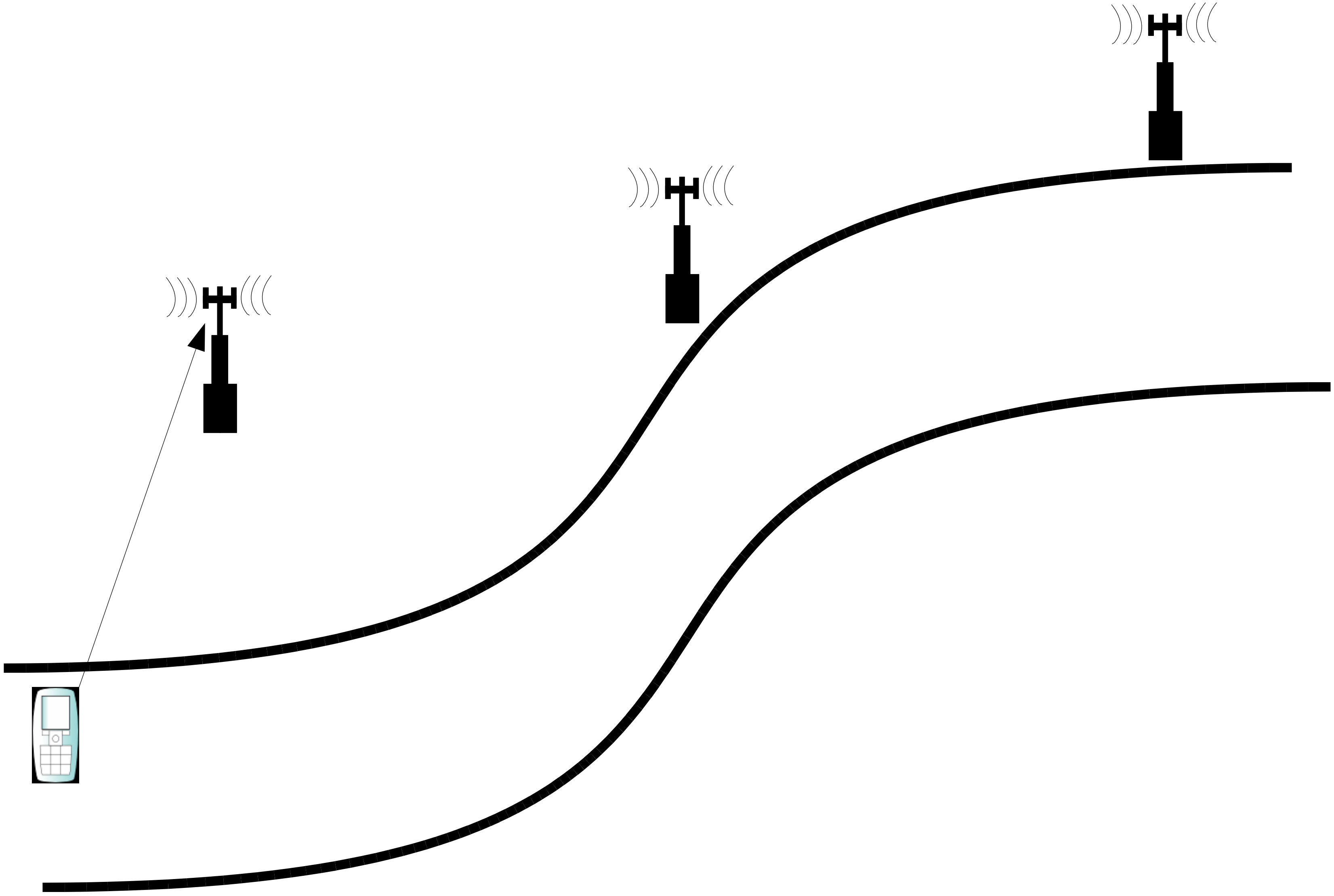


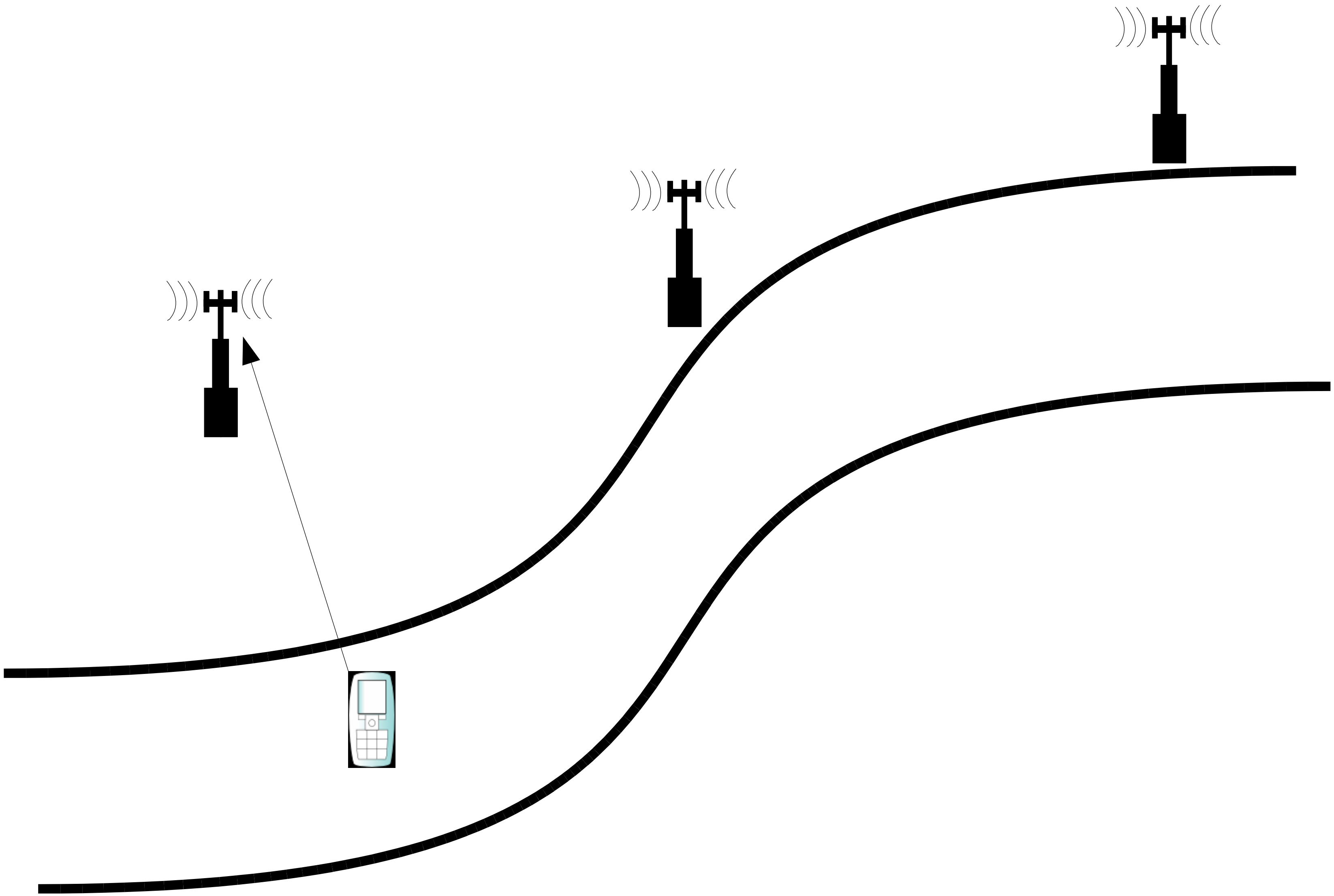
Improved BRKGA:

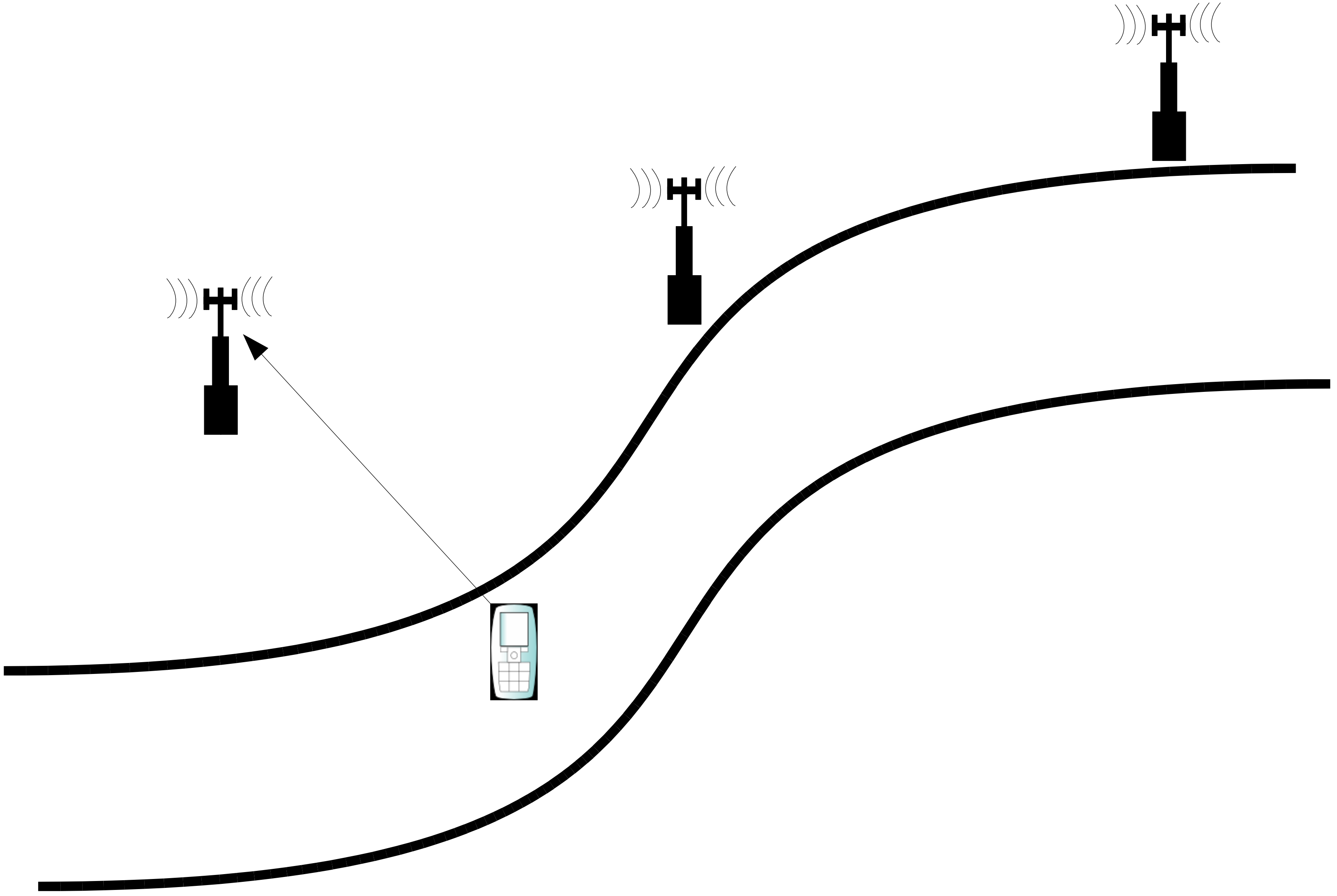
Finds solutions faster

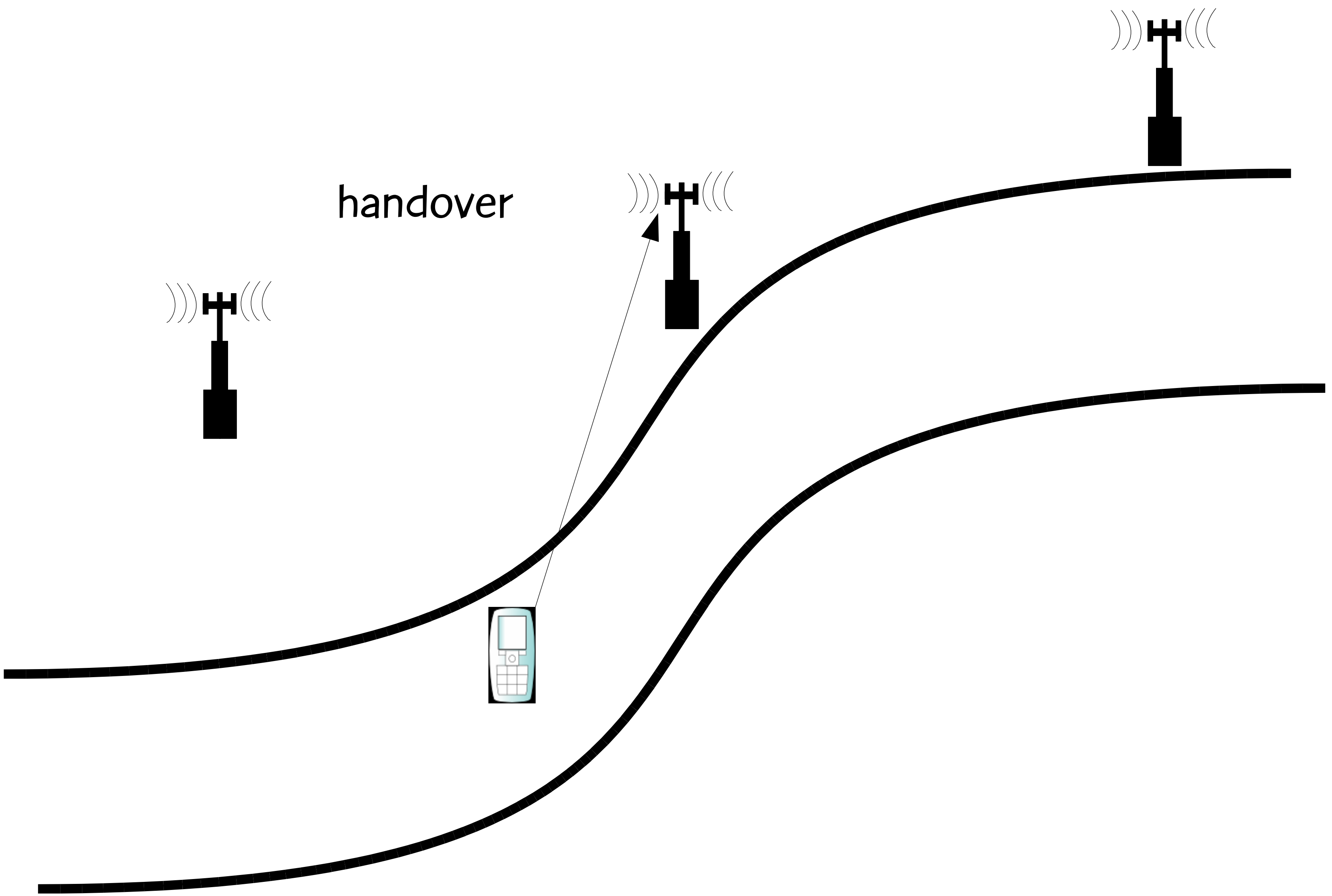
Finds better solutions

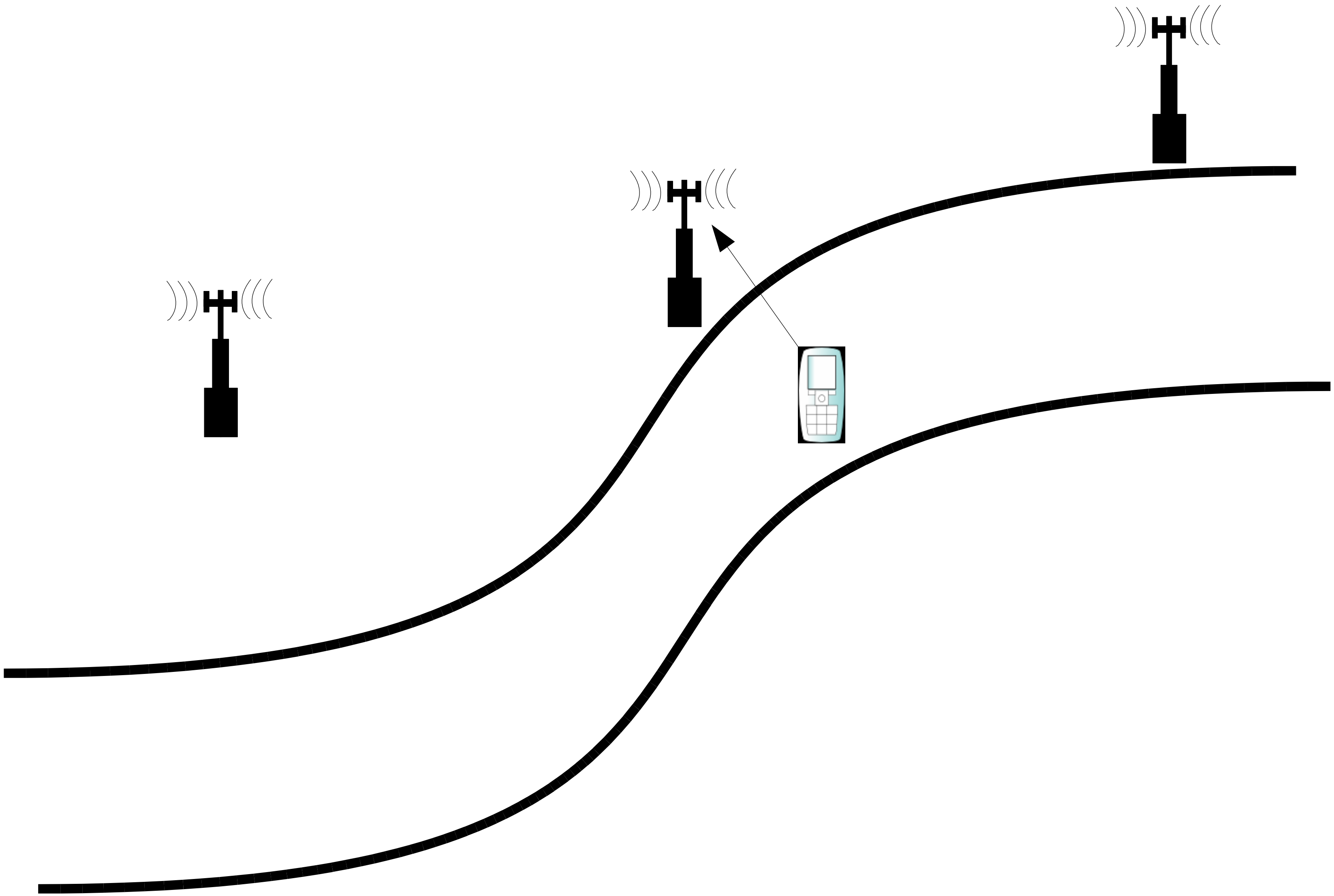
Handover minimization in mobility networks



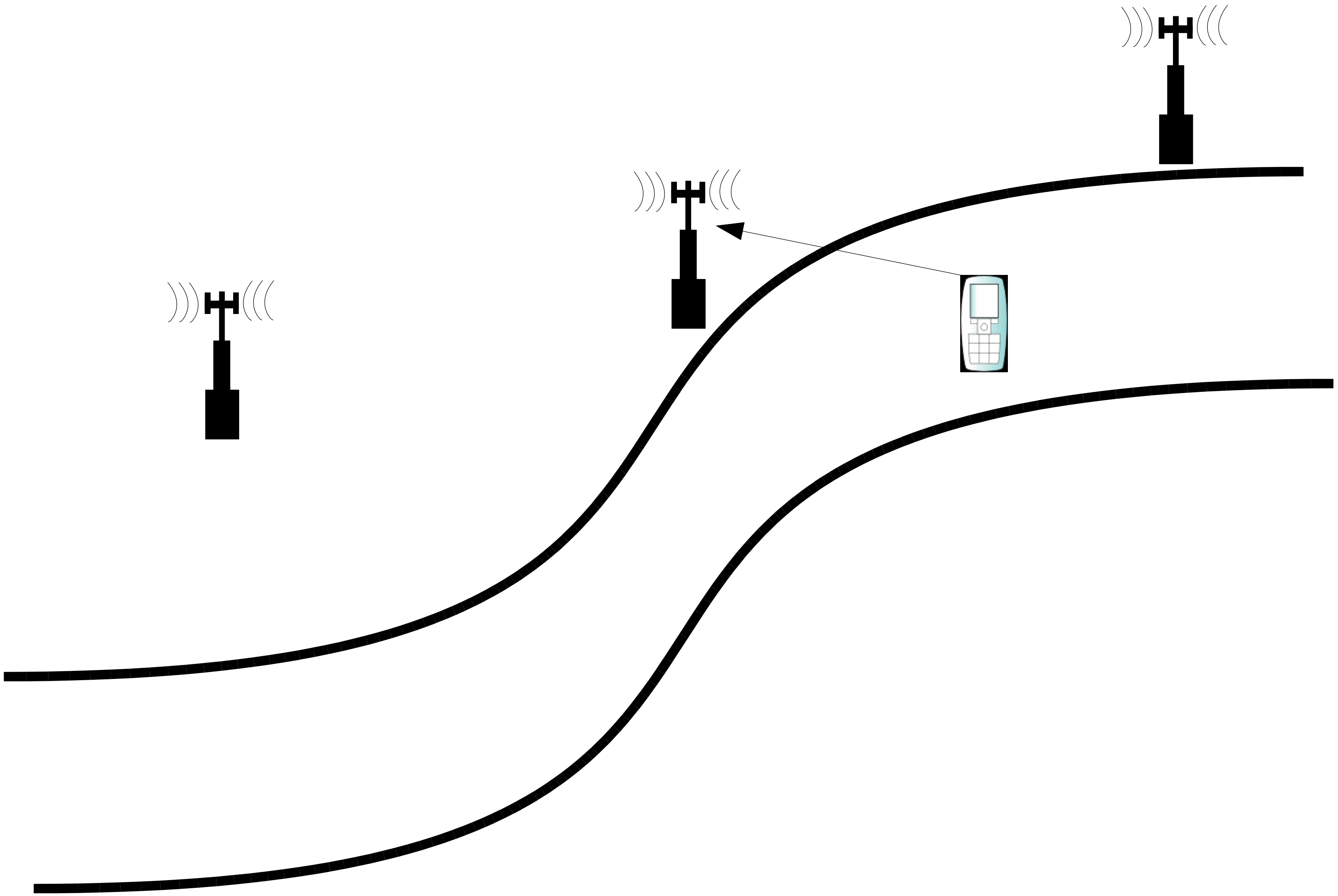


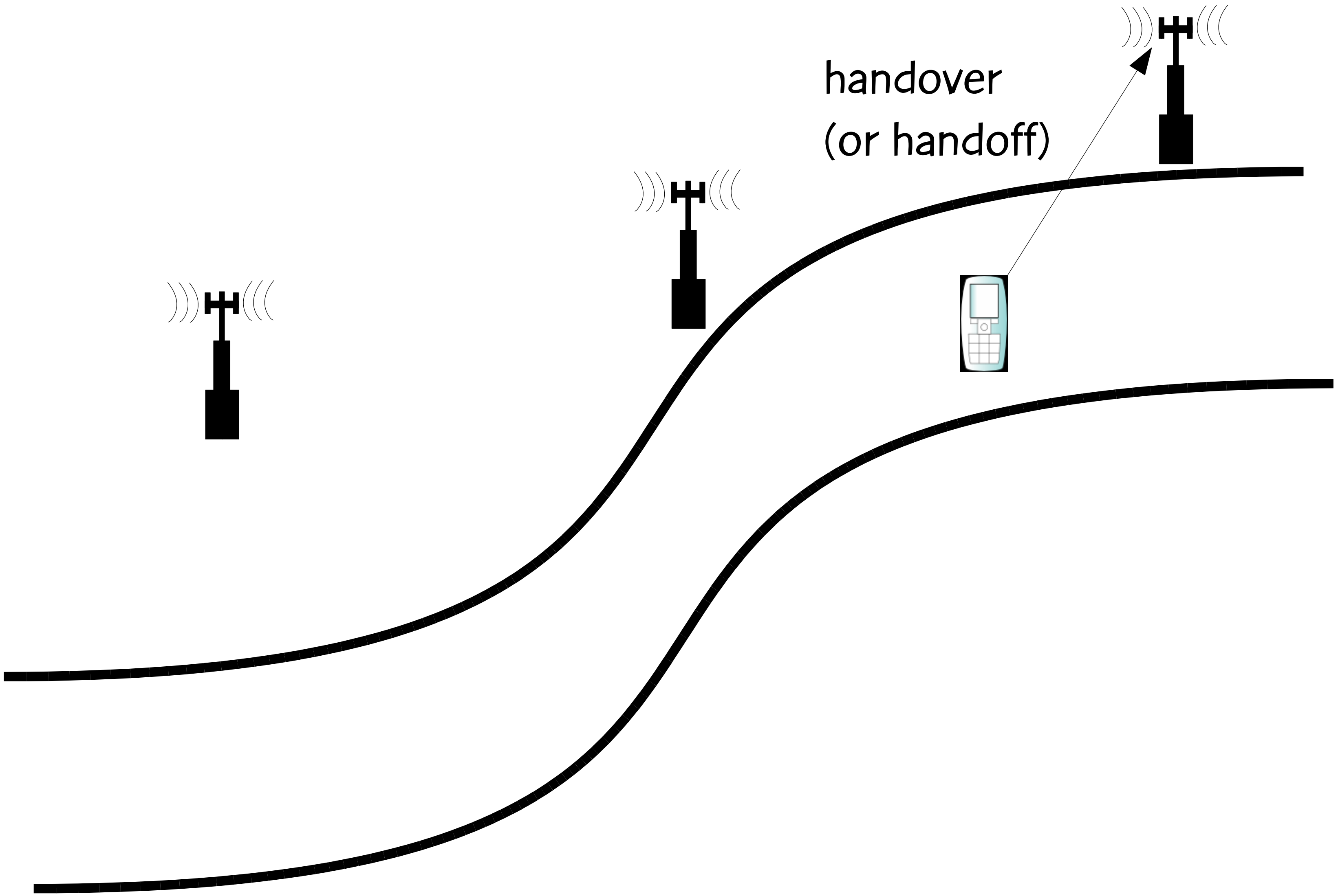


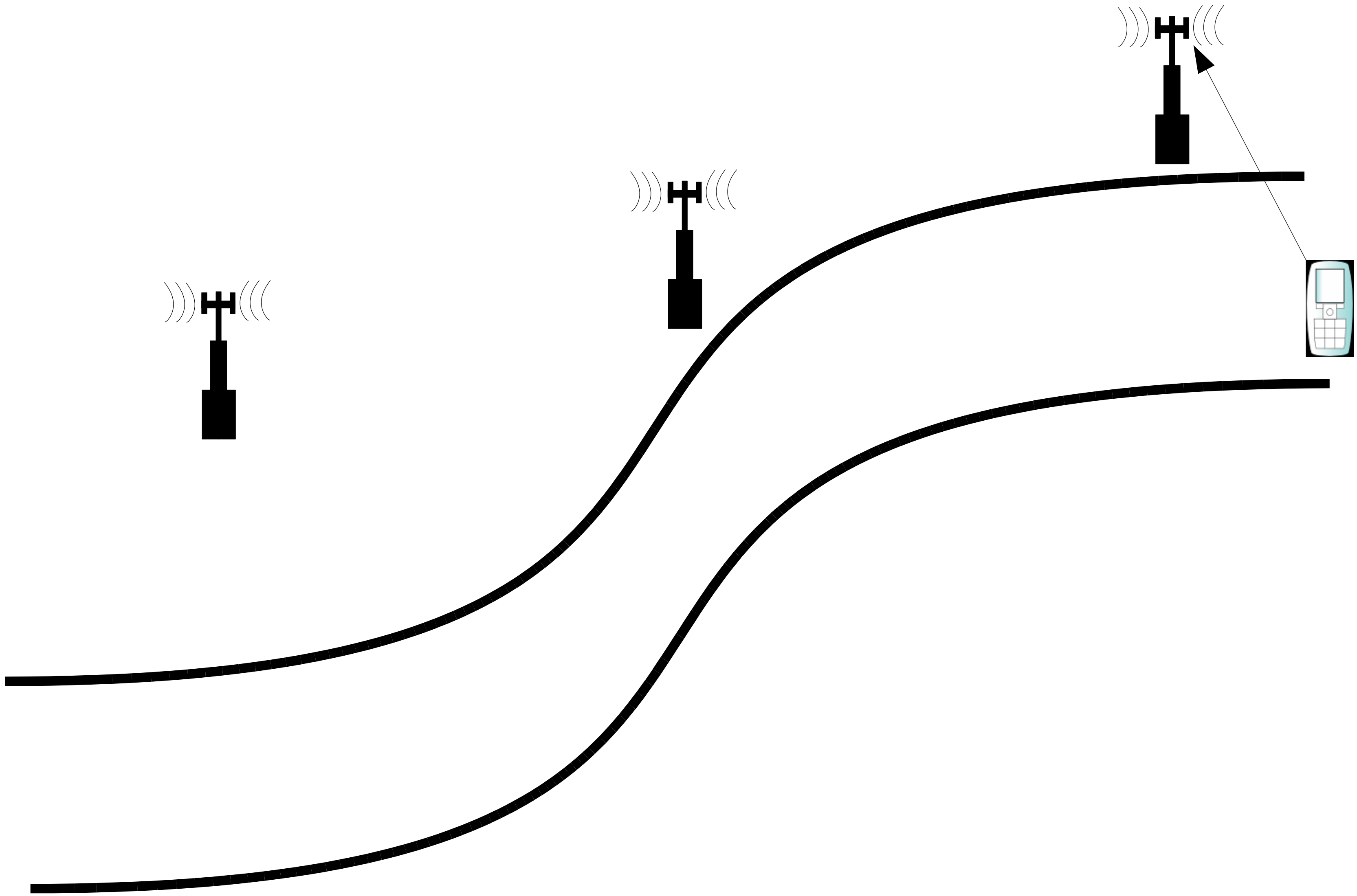

handover

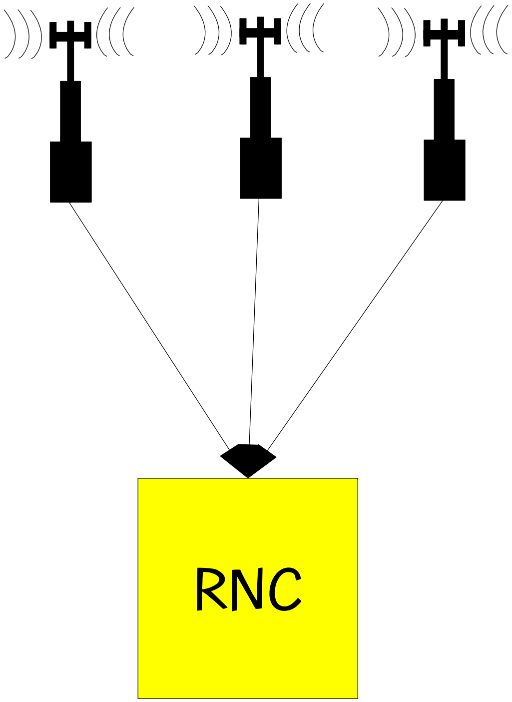


at&t
Your world. Delivered.

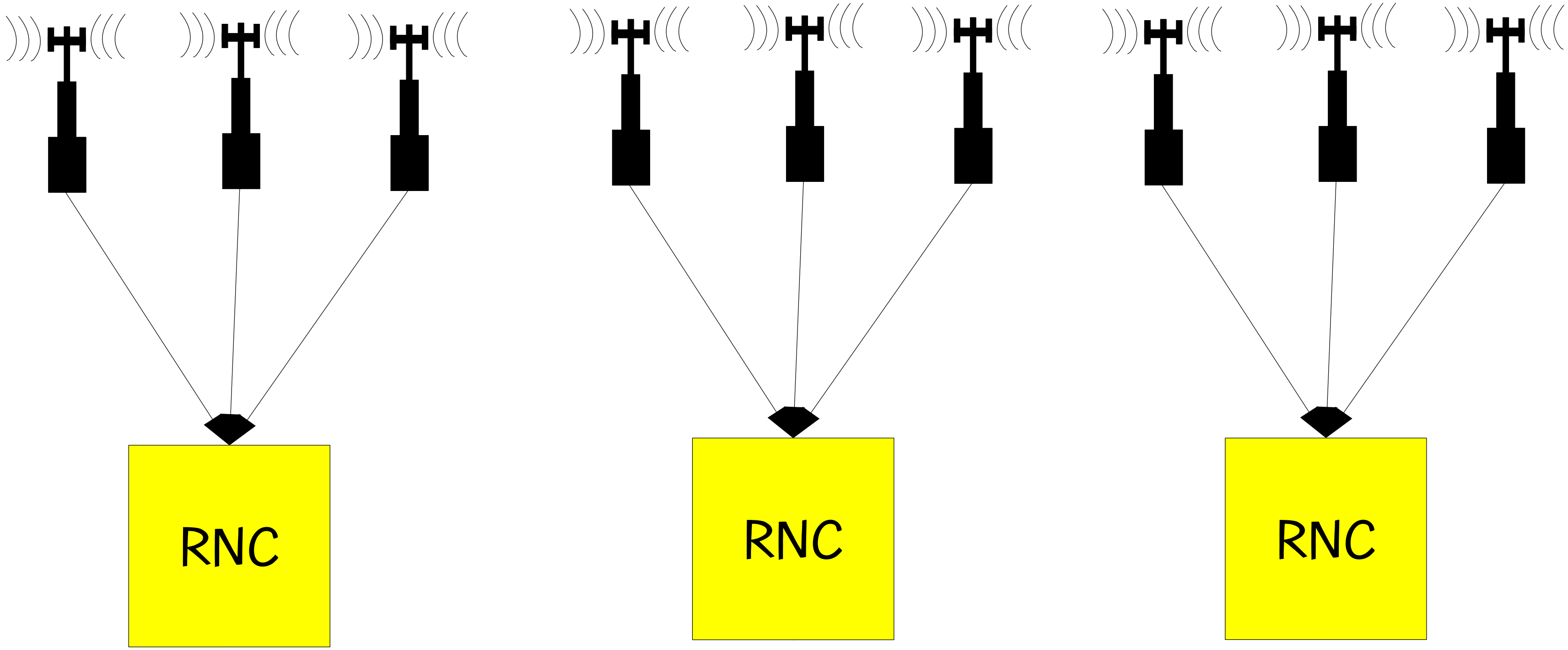



handover
(or handoff)

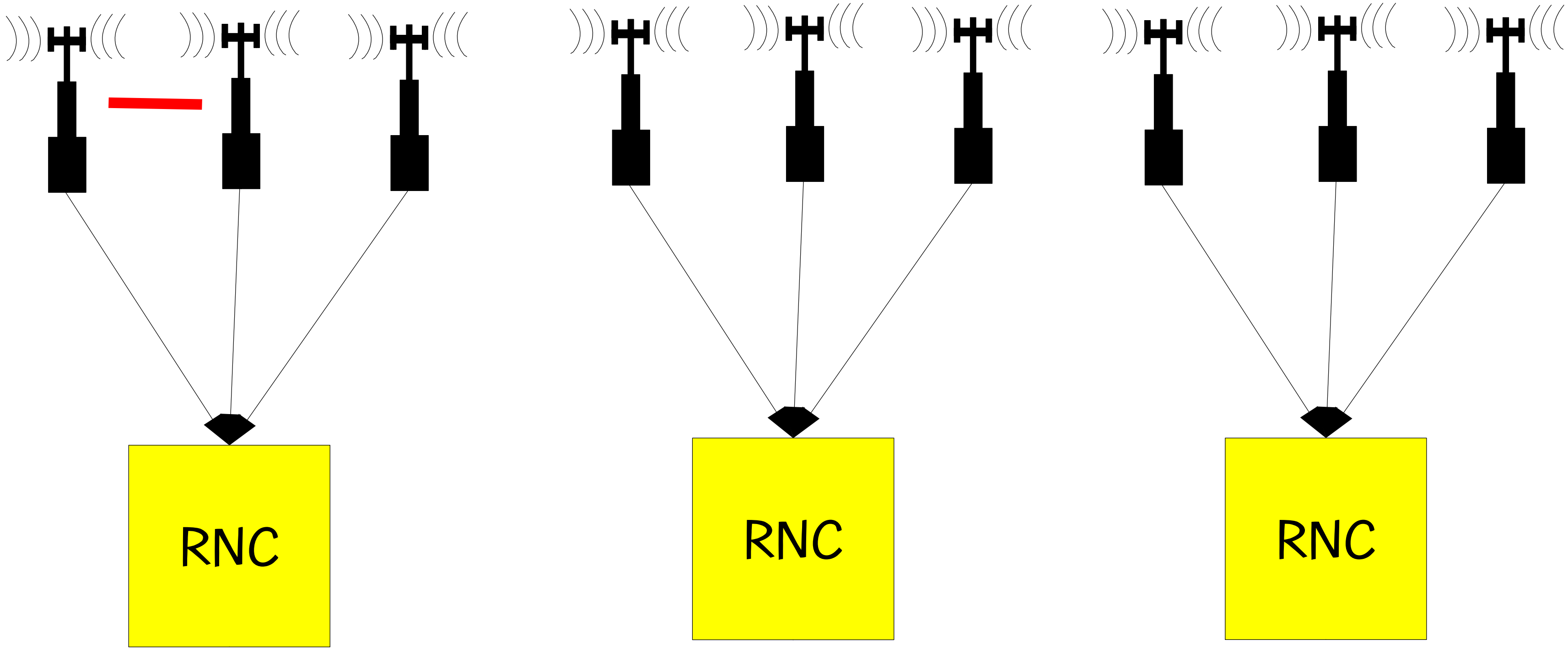




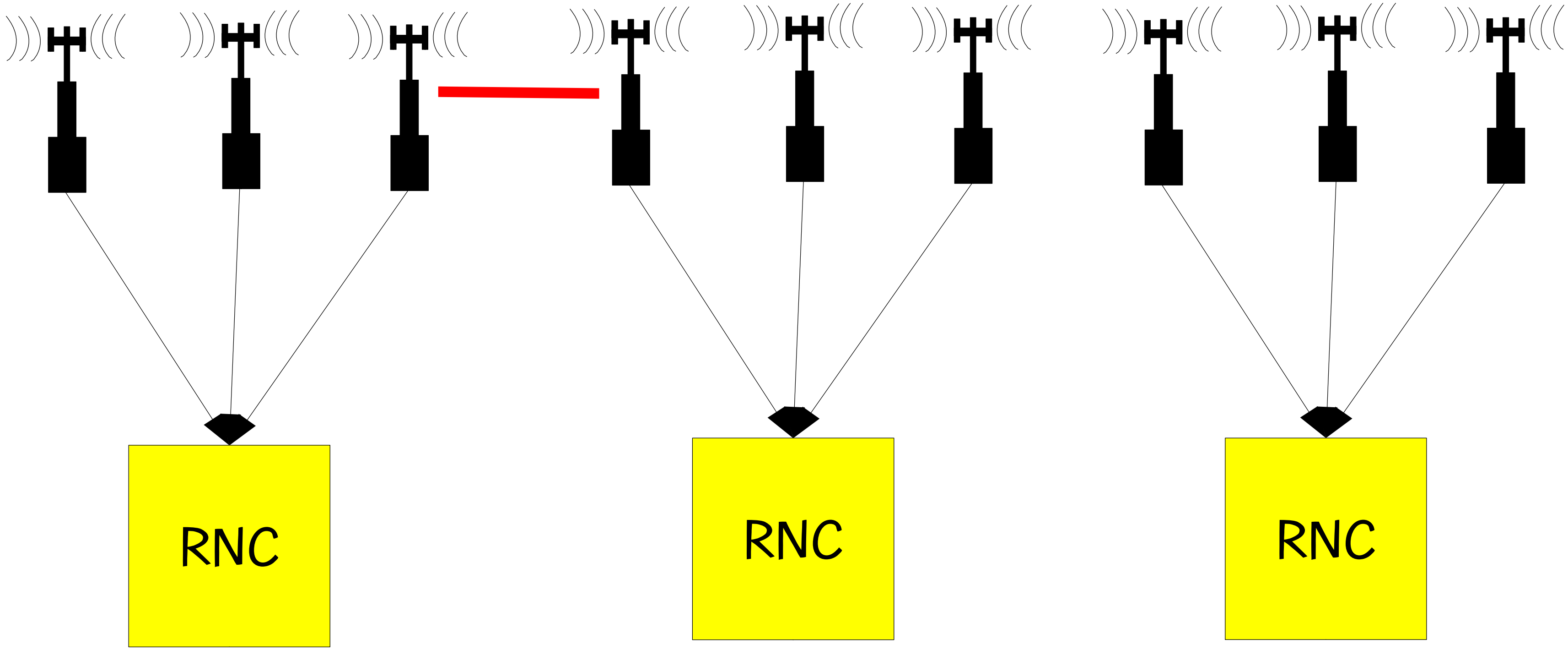
- Each cell tower has associated with it an amount of traffic.
- Each cell tower is connected to a Radio Network Controller (RNC).
- Each RNC can have one or more cell towers connected to it.
- Each RNC can handle a given amount of traffic ... this limits the subsets of cell towers that can be connected to it.
- An RNC controls the cell towers connected to it.



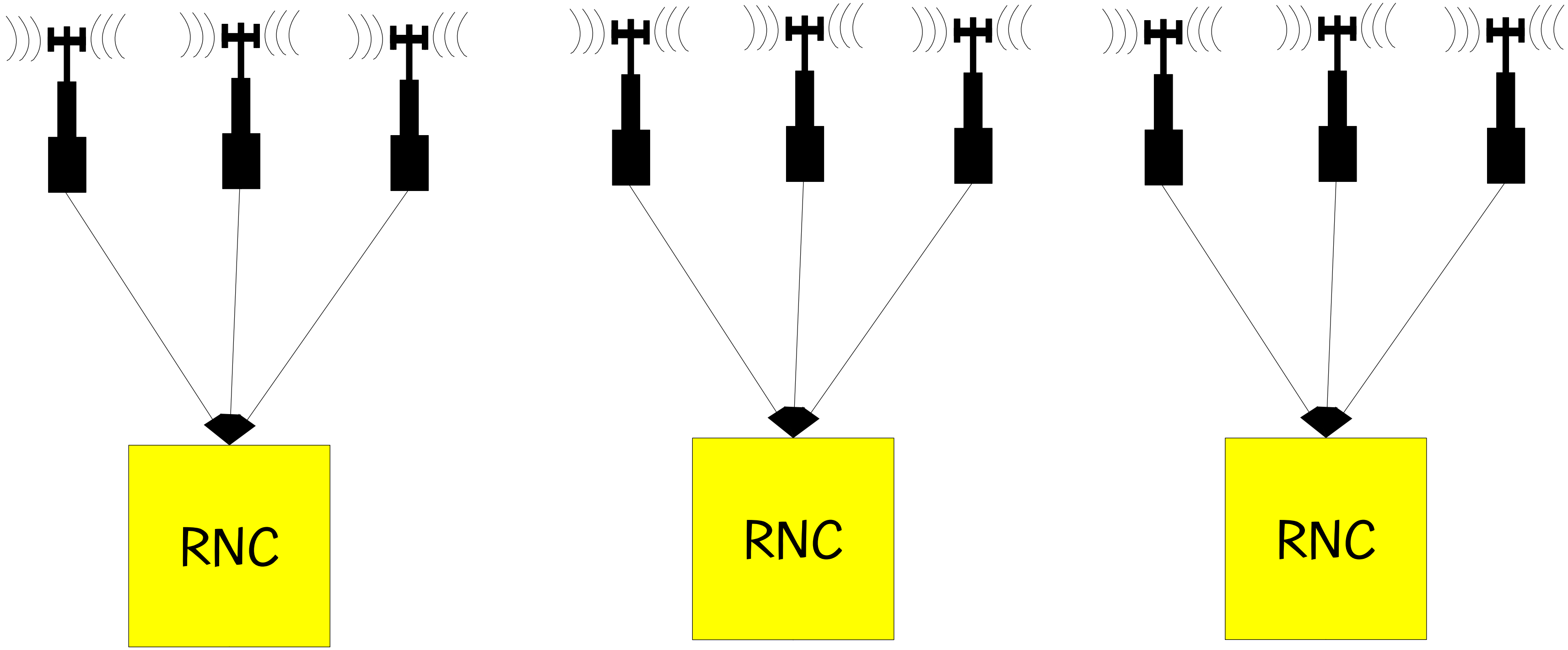
- Handovers can occur between towers



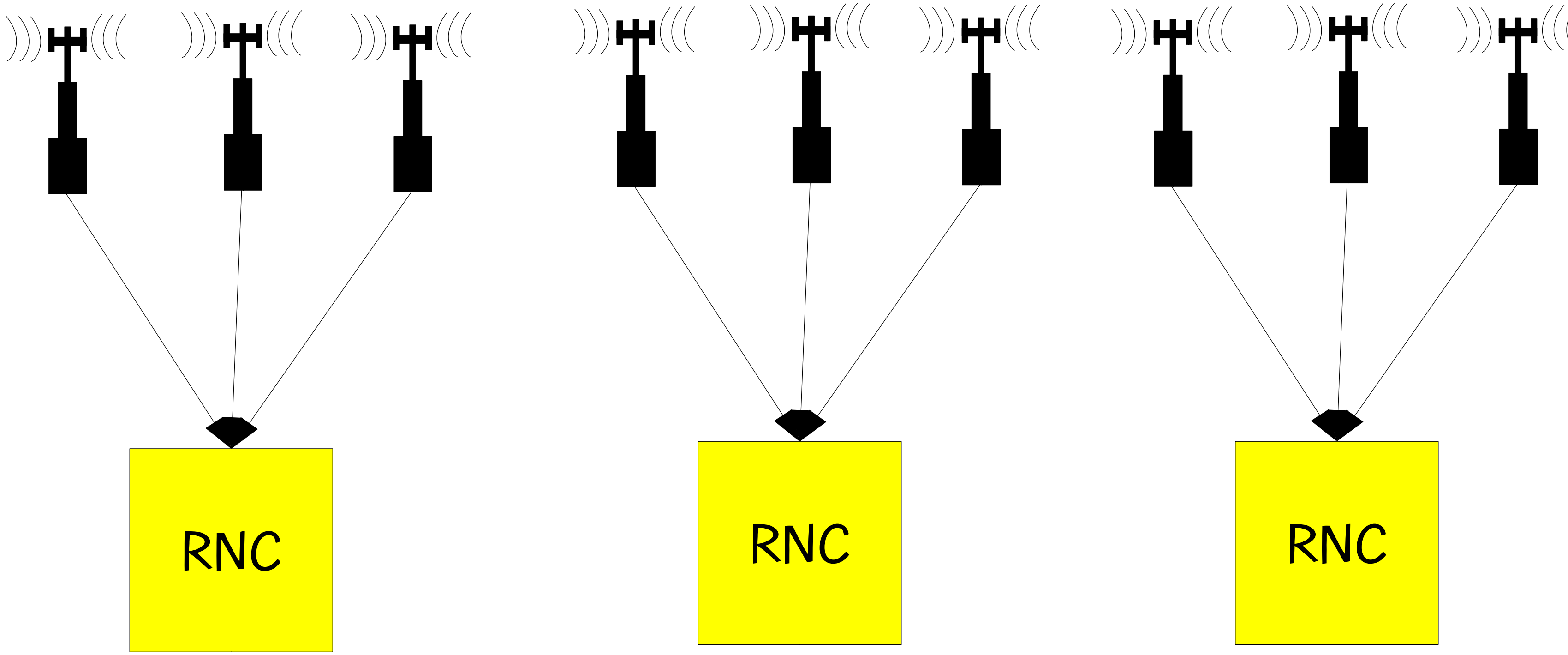
- Handovers can occur between towers
  - connected to the same RNC



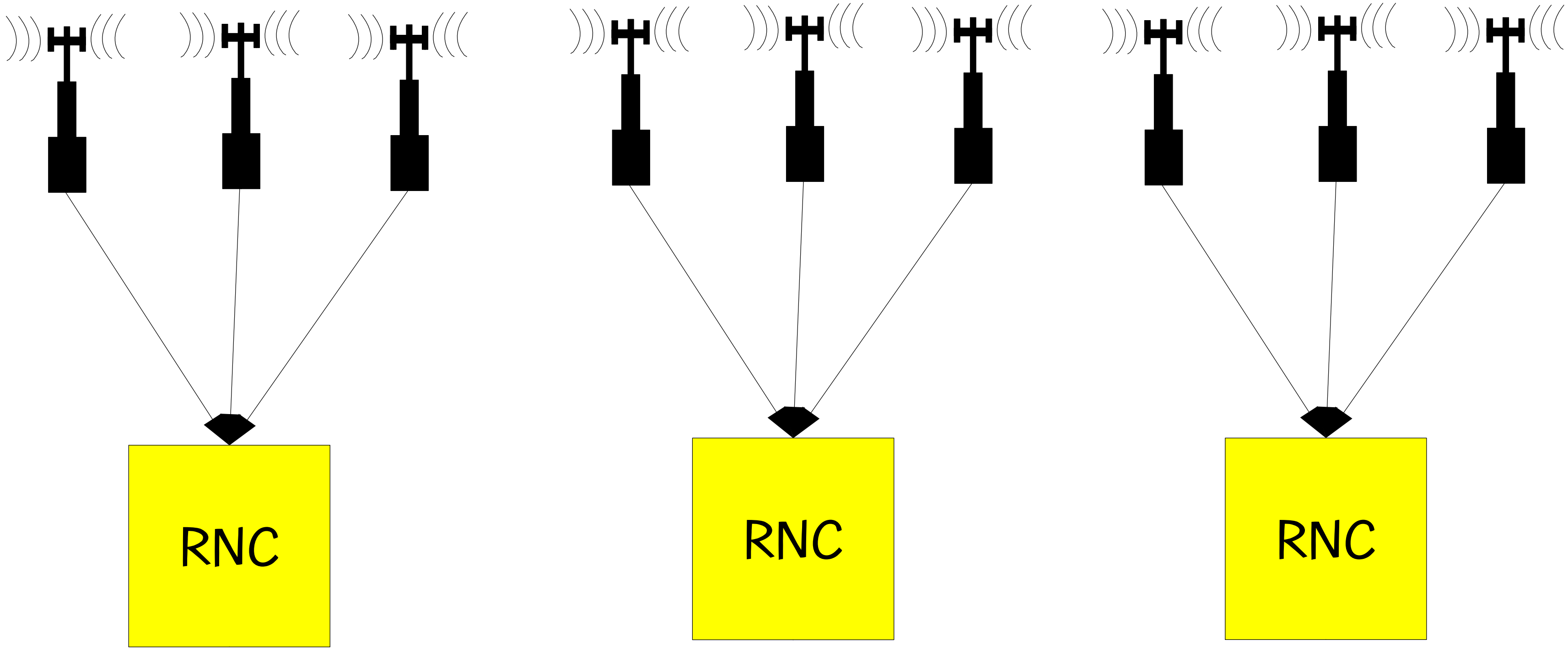
- Handovers can occur between towers
  - connected to the same RNC
  - connected to different RNCs



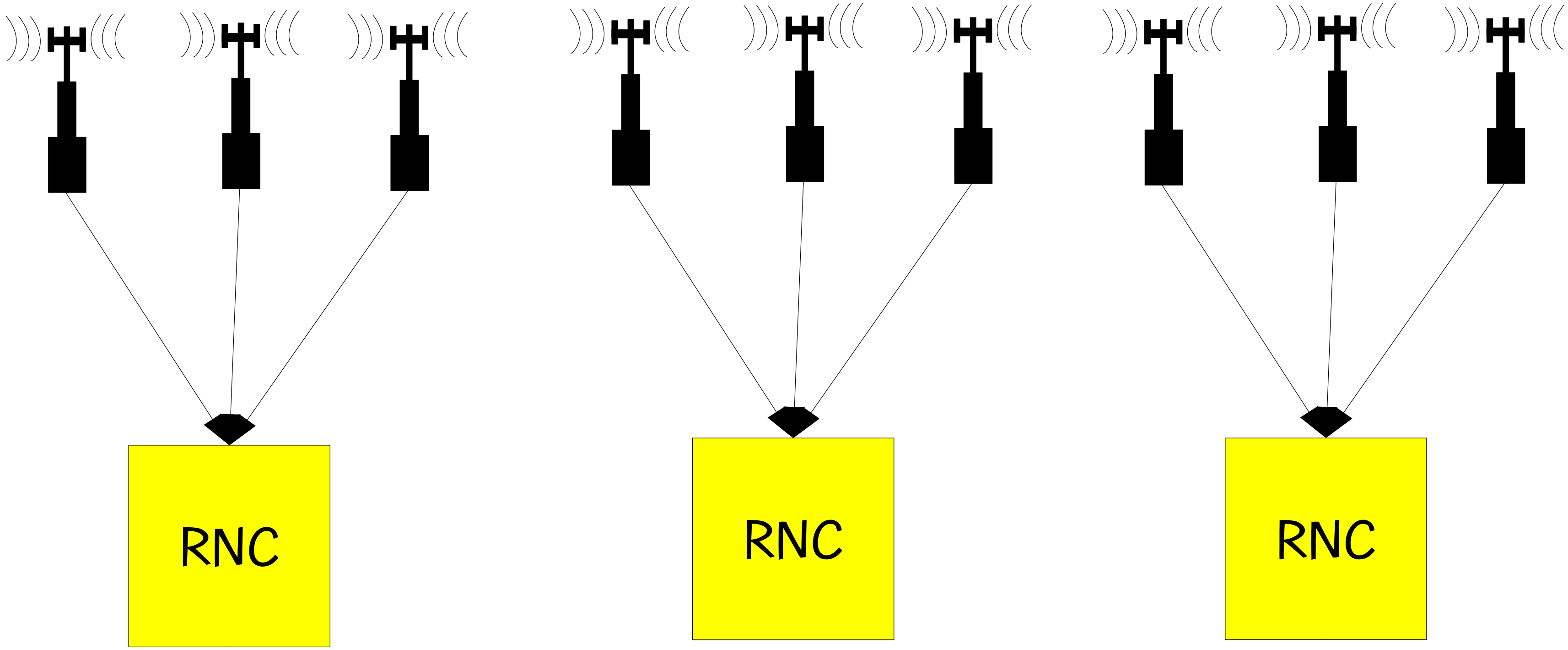
- Handovers between towers connected to different RNCs tend to fail more often than handovers between towers connected to the same RNC.
- Handover failure results in dropped call!



- If we minimize the number of handovers between towers connected to different RNCs we may be able to reduce the number of dropped calls.



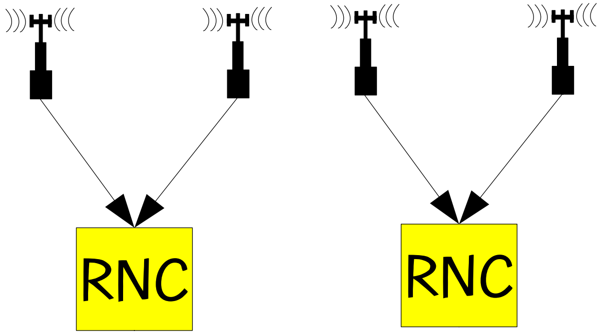
- HANDOVER MINIMIZATION: Assign towers to RNCs such that RNC capacity is not violated and number of handovers between towers assigned to different RNCs is minimized.



- HANDOVER MINIMIZATION: Assign towers to RNCs such that RNC capacity is not violated and number of handovers between towers assigned to different RNCs is minimized.

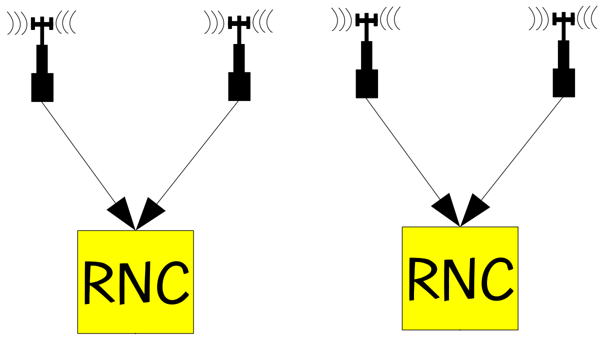
Node-capacitated graph partitioning problem

Example

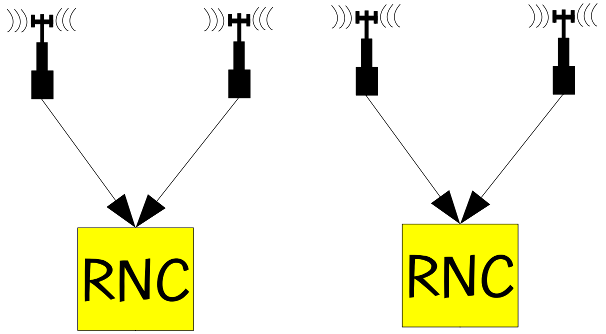


- 4 towers: t(1) = 25; t(2) = 15; t(3) = 35; t(4) = 25
- 2 RNCs: c(1) = 50; c(2) = 60
- Handover matrix:

	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

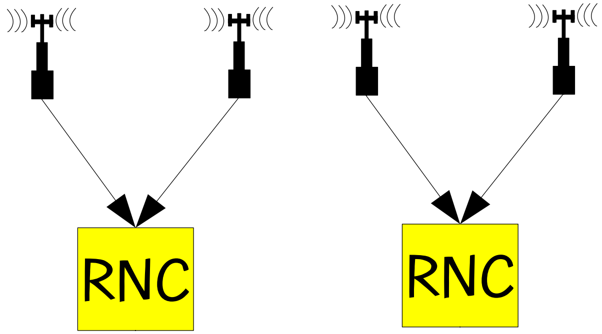


- 4 towers: t(1) = 25; t(2) = 15; t(3) = 35; t(4) = 25
- 2 RNCs: c(1) = 50; c(2) = 60
- Given this traffic profile and RNC capacities the feasible configurations are:
  - RNC(1): { 1, 2 }; RNC(2): { 3, 4 }
  - RNC(1): { 2, 3 }; RNC(2): { 1, 4 }
  - RNC(1): { 2, 4 }; RNC(2): { 1, 3 }
  - RNC(1): { 1, 4 }; RNC(2): { 2, 3 }



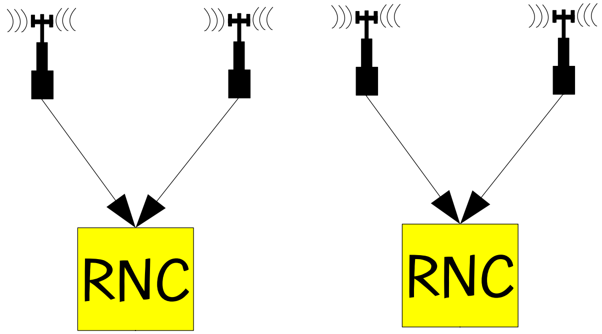
	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

- Total handover for each configuration:



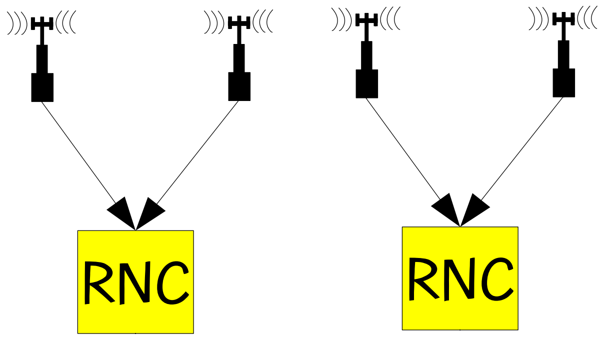
	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

- Total handover for each configuration:
  - RNC(1): { 1, 2 }; RNC(2): { 3, 4 }: h(1,3) + h(1,4) + h(2,3) + h(2,4) = 10 + 0 + 200 + 50 = 260



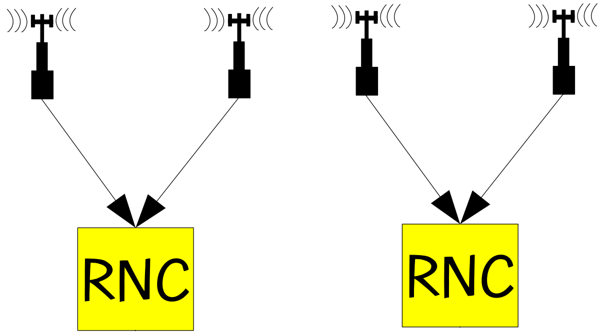
	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

- Total handover for each configuration:
  - RNC(1): { 1, 2 }; RNC(2): { 3, 4 }: h(1,3) + h(1,4) + h(2,3) + h(2,4) = 10 + 0 + 200 + 50 = 260
  - RNC(1): { 2, 3 }; RNC(2): { 1, 4 }: h(2,1) + h(2,4) + h(3,1) + h(3,4) = 100 + 50 + 10 + 500 = 660



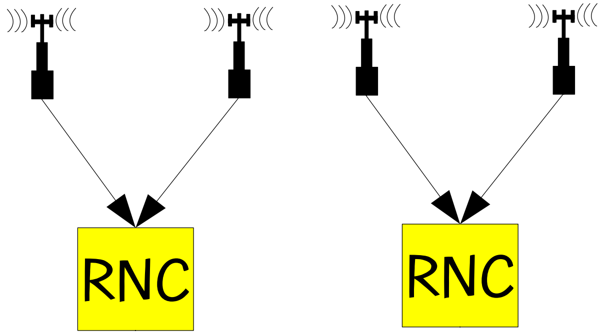
	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

- Total handover for each configuration:
  - RNC(1): { 1, 2 }; RNC(2): { 3, 4 }: h(1,3) + h(1,4) + h(2,3) + h(2,4) = 10 + 0 + 200 + 50 = 260
  - RNC(1): { 2, 3 }; RNC(2): { 1, 4 }: h(2,1) + h(2,4) + h(3,1) + h(3,4) = 100 + 50 + 10 + 500 = 660
  - RNC(1): { 2, 4 }; RNC(2): { 1, 3 }: h(2,1) + h(2,3) + h(4,1) + h(4,3) = 100 + 200 + 0 + 500 = 800



	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

- Total handover for each configuration:
  - RNC(1): { 1, 2 }; RNC(2): { 3, 4 }: h(1,3) + h(1,4) + h(2,3) + h(2,4) = 10 + 0 + 200 + 50 = 260
  - RNC(1): { 2, 3 }; RNC(2): { 1, 4 }: h(2,1) + h(2,4) + h(3,1) + h(3,4) = 100 + 50 + 10 + 500 = 660
  - RNC(1): { 2, 4 }; RNC(2): { 1, 3 }: h(2,1) + h(2,3) + h(4,1) + h(4,3) = 100 + 200 + 0 + 500 = 800
  - RNC(1): { 1, 4 }; RNC(2): { 2, 3 }: h(1,2) + h(1,3) + h(4,2) + h(4,3) = 100 + 10 + 50 + 500 = 660

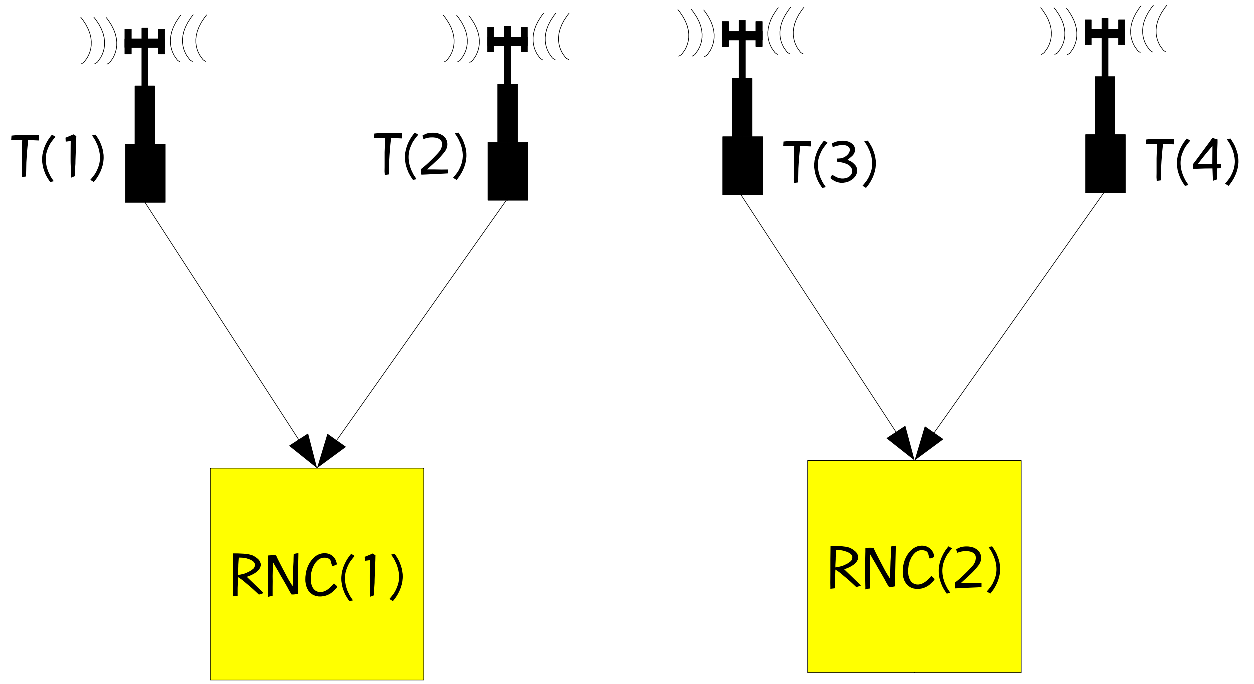


	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

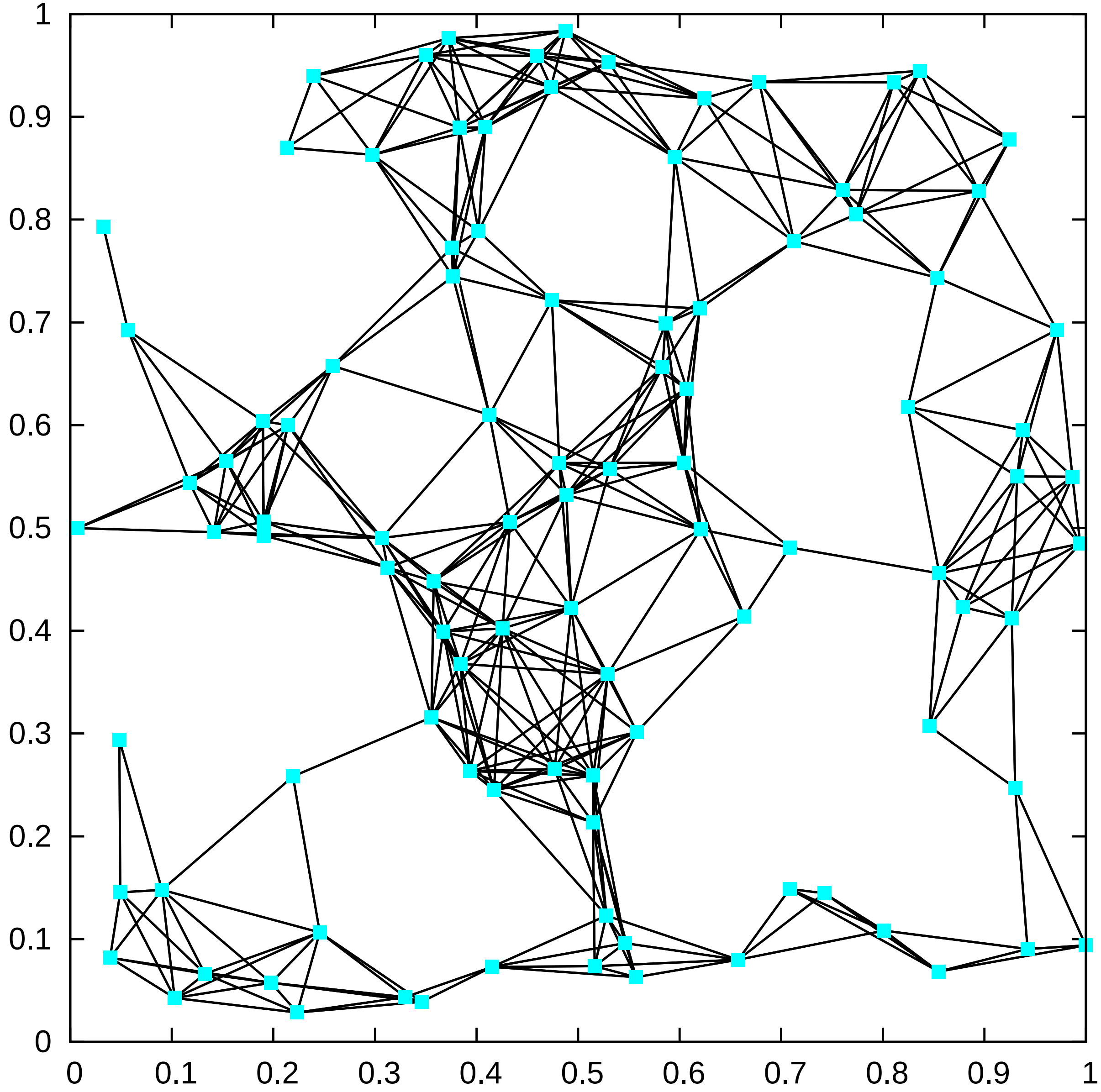
- Total handover for each configuration:
  - RNC(1): { 1, 2 }; RNC(2): { 3, 4 }: h(1,3) + h(1,4) + h(2,3) + h(2,4) = 10 + 0 + 200 + 50 = **260**
  - RNC(1): { 2, 3 }; RNC(2): { 1, 4 }: h(2,1) + h(2,4) + h(3,1) + h(3,4) = 100 + 50 + 10 + 500 = 660
  - RNC(1): { 2, 4 }; RNC(2): { 1, 3 }: h(2,1) + h(2,3) + h(4,1) + h(4,3) = 100 + 200 + 0 + 500 = 800
  - RNC(1): { 1, 4 }; RNC(2): { 2, 3 }: h(1,2) + h(1,3) + h(4,2) + h(4,3) = 100 + 10 + 50 + 500 = 660

	1	2	3	4
1	0	100	10	0
2	100	0	200	50
3	10	200	0	500
4	0	50	500	0

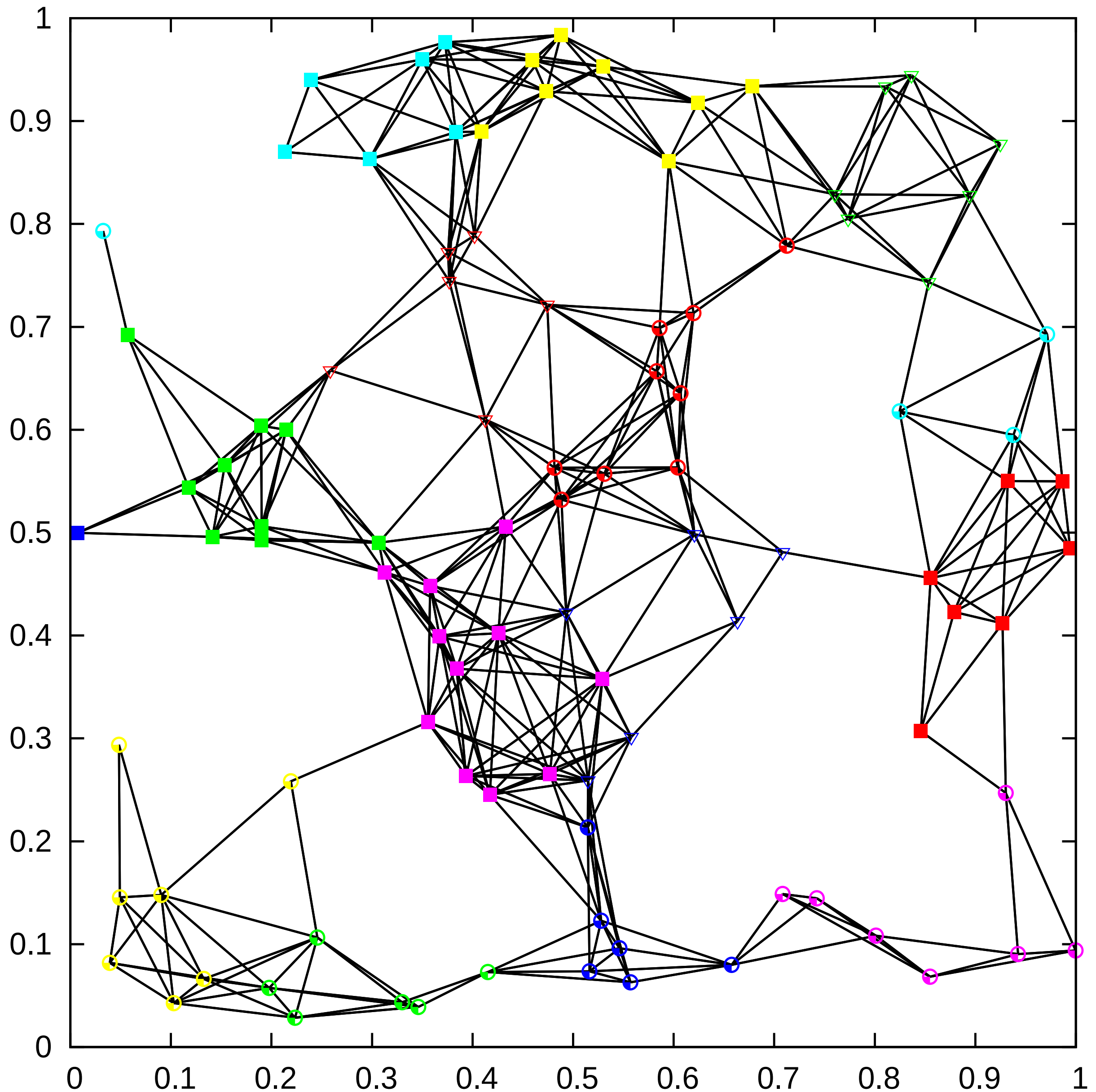
Optimal configuration:



G=(T,E) Nodeset T are the towers; Edgeset: (i,j)∈E iff h(i,j)+h(j,i) > 0



Tower are assigned to RNCs indicated by distinct colors/shapes



Mixed integer programming formulation

- T is the set of towers
- R is the set of RNCs
- $x_{e,k} = 1$ if edge e =(i,j) has both endpoints in RNC k
- $y_{i,k} = 1$ if tower i is assigned to RNC k

Mixed integer programming formulation

Each tower can only be assigned to one RNC:

$$\text{sum}_{\{k \in R\}} y_{i,k} = 1, \text{ for all } i \in T$$

Mixed integer programming formulation

Each e=(i,j) cannot be in RNC k if either of its endpoints is not assigned to RNC k:

$$x_{e,k} \leq y_{i,k}, \text{ for all } e=(i,j) \in E, k \in R$$

$$x_{e,k} \leq y_{j,k}, \text{ for all } e=(i,j) \in E, k \in R$$

$$x_{e,k} \geq y_{i,k} + y_{j,k} - 1, \text{ for all } e=(i,j) \in E, k \in R$$

Mixed integer programming formulation

Each RNC k can only accommodate c_k units of traffic:

$$\text{sum}_{\{i \in T\}} t_i y_{i,k} \leq c_k, \text{ for all } k \in R$$

Mixed integer programming formulation

Minimize handover between towers assigned to different RNCs is equivalent to maximize handover between towers assigned to the same RNC.

Objective function:

$$\max \{ \text{sum}_{\{k \in R\}} \{ \text{sum}_{\{e=(i,j) \in E\}} h(i,j)\ x_{e,k} \} \}$$

at&t Your world. Delivered.

CPLEX MIP solver

Towers	RNCs	BKS	CPLEX	time (s)
20	10	7602	7602	18.80
30	15	18266	18266	25911.00
40	15	29700	29700	101259.91
100	15	19000	49270	1 day
100	25	36412	58637	1 day
100	50	60922	70740	1 day

CPLEX MIP solver

Towers	RNCs	BKS	CPLEX	time (s)
20	10	7602	7602	18.80
30	15	18266	18266	25911.00
40	15	29700	29700	101259.91
100	15	19000	49270	1 day
100	25	36412	58637	1 day
100	50	60922	70740	1 day

We would like to solve instances with 1000 towers.

CPLEX MIP solver

Towers	RNCs	BKS	CPLEX	time (s)
20	10	7602	7602	18.80
30	15	18266	18266	25911.00
40	15	29700	29700	101259.91
100	15	19000	49270	1 day
100	25	36412	58637	1 day
100	50	60922	70740	1 day

We would like to solve instances with 1000 towers.

Need heuristics!

A simple BRKGA for HMP

Encoding

Each solution is encoded as a vector of $|T|$ random keys, where $|T|$ is the number of towers

Decoding

Decoder takes input a vector of |T| random keys and outputs a tower-to-RNC assignment:

1) sort vector resulting in ordering of towers

2) scan towers in order ...

- place tower in RNC with available capacity with which the tower has greatest number of handovers with other towers already assigned to RNC
- if RNC with available capacity does not exist, open a new artificial RNC with capacity $\max \{ c_i \mid i \in \text{open RNCs} \}$

3) apply tower move local search to produce local minimum

Another BRKGA for HMP

Encoding

Each solution is encoded as a vector of $2\ |T|$ random keys, where $|T|$ is the number of towers

Decoding

Decoder takes input a vector of 2 |T| random keys and outputs a tower-to-RNC assignment:

1) sort first |T| keys resulting in ordering of towers

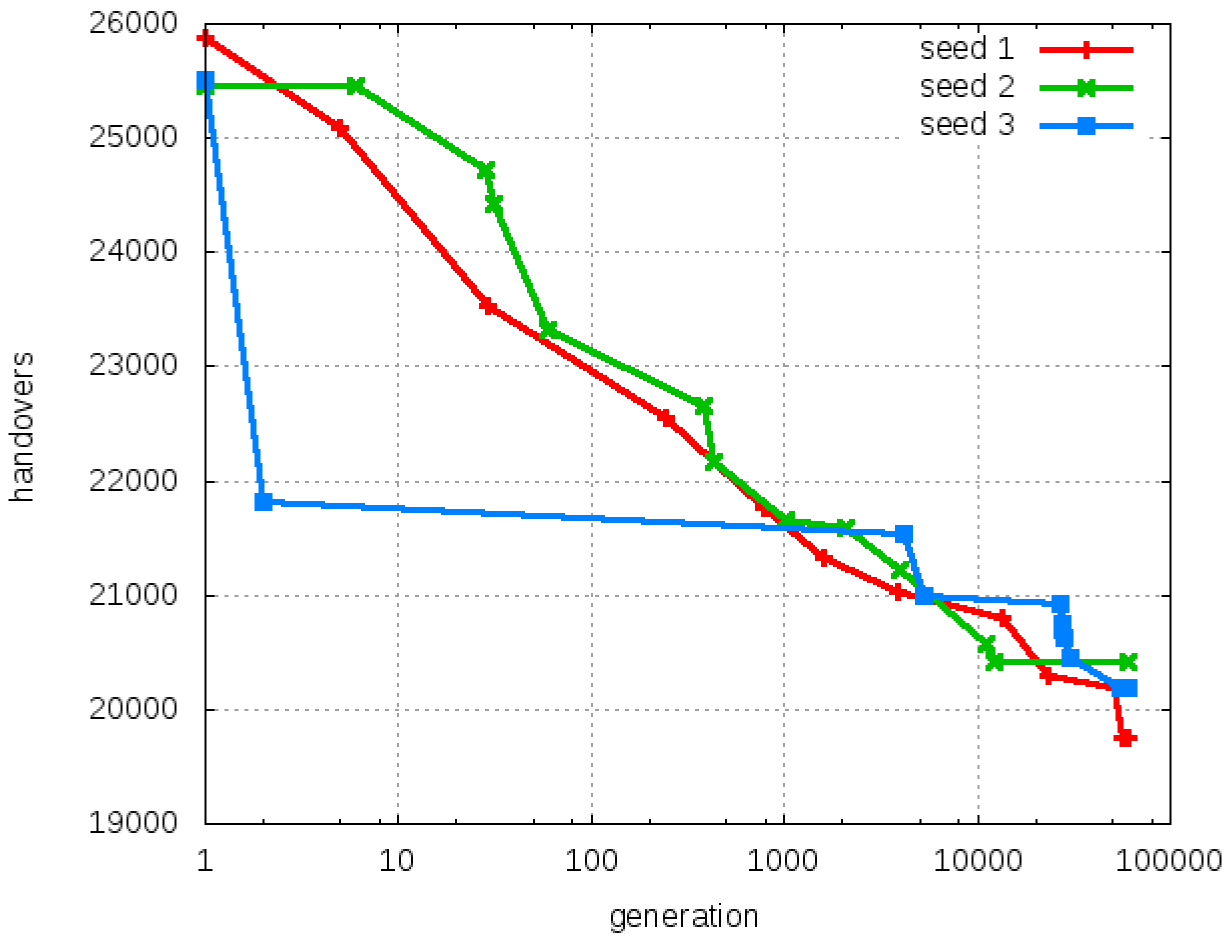
2) scan towers in order ...

- place tower in RNC with available capacity as indicated by mapping [0,1) to [1, 2, .., |RNCs|] from second |T| keys
- scan unassigned towers in order and place them in RNC with available capacity maximizing handover count with tower assigned there
- if RNC with available capacity does not exist, assign tower to RNC with maximum handover count w.r.t. to tower

3) apply tower move local search to produce local minimum

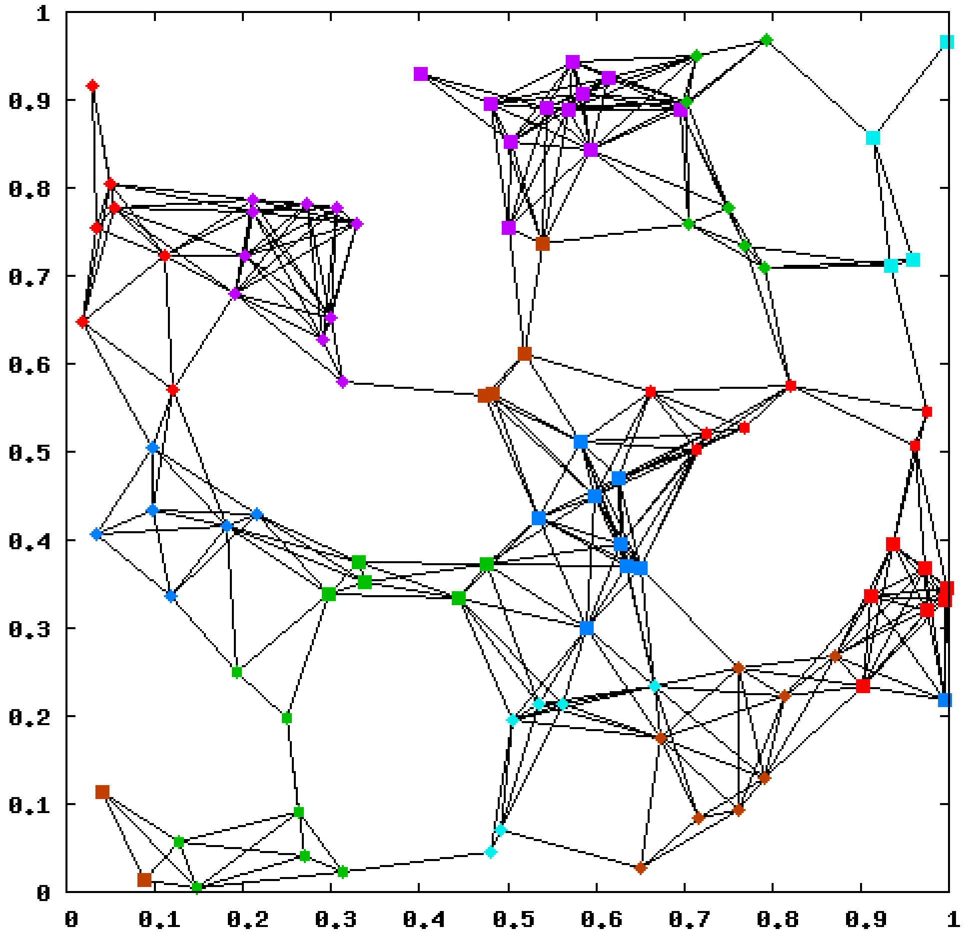
Experiments with BRKGA-1 for HMP

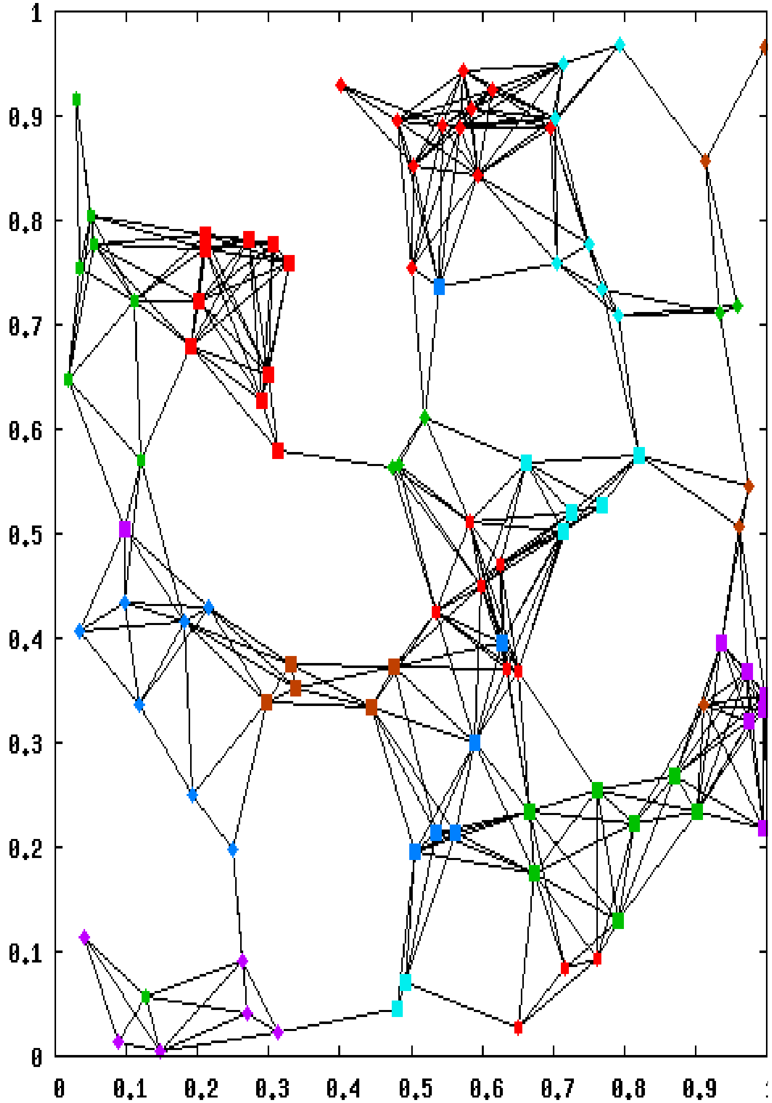
BRKGA: 100 towers : 14 RNCs



BRKGA: 100 towers : 14 RNCs

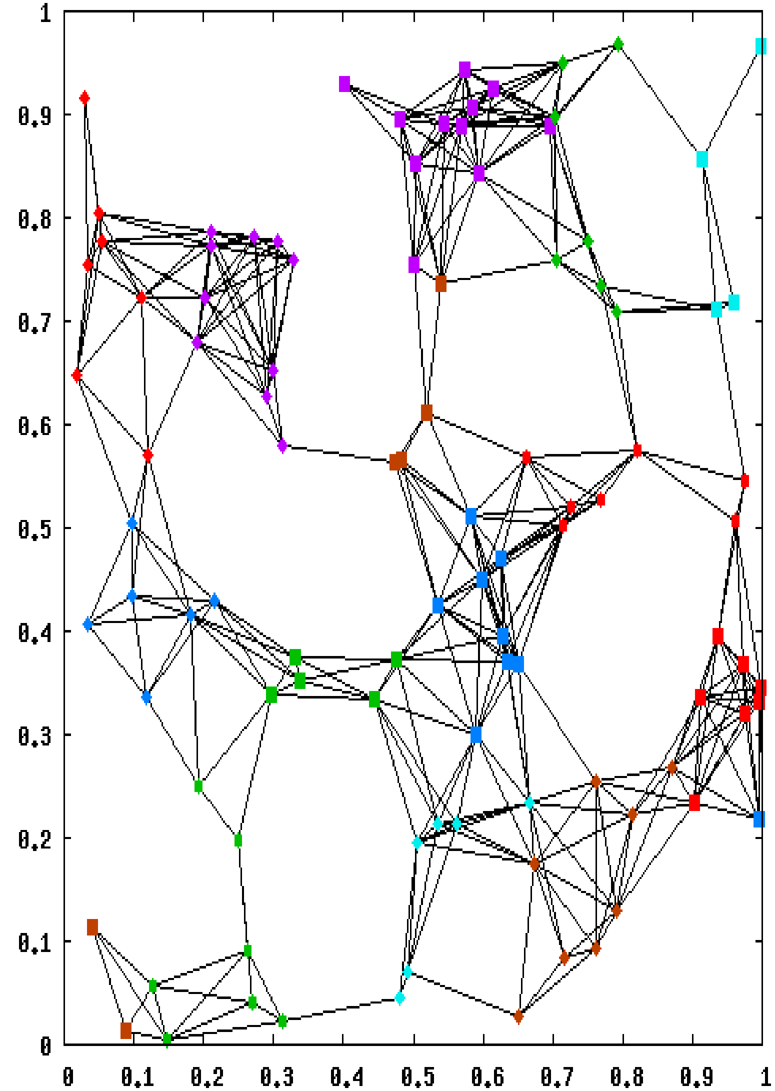
Generation: 56324
Handovers: 19750





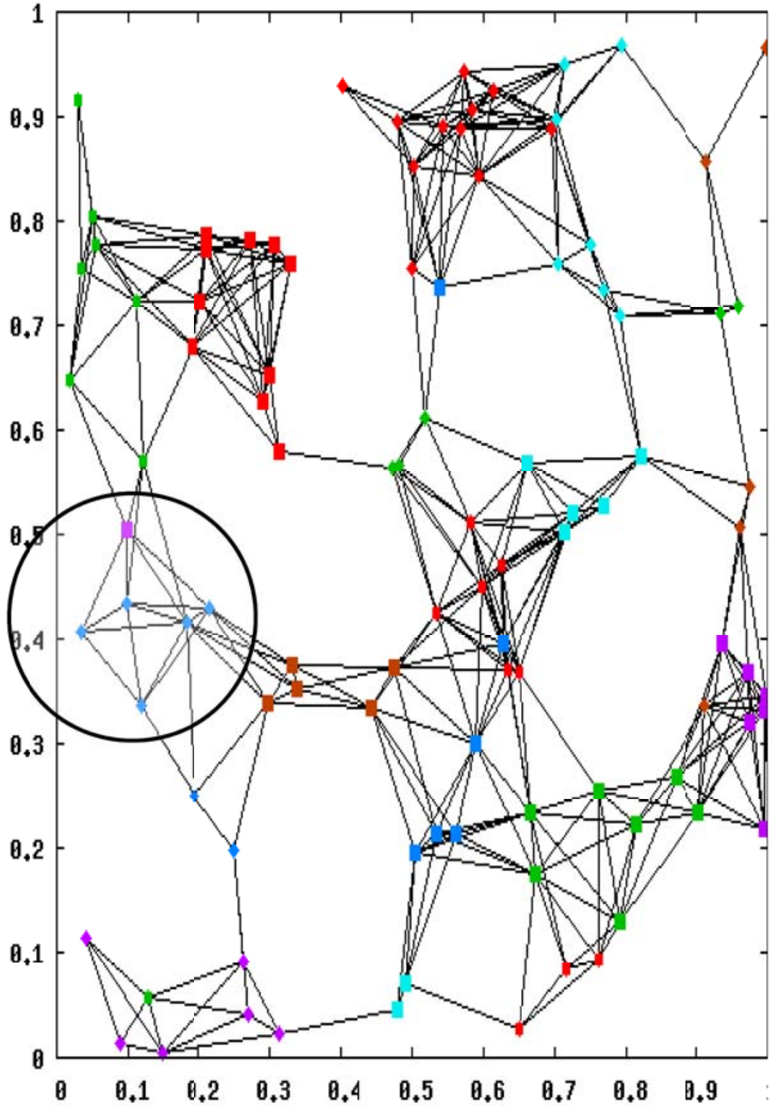
Generation: 1

Handovers: 25872



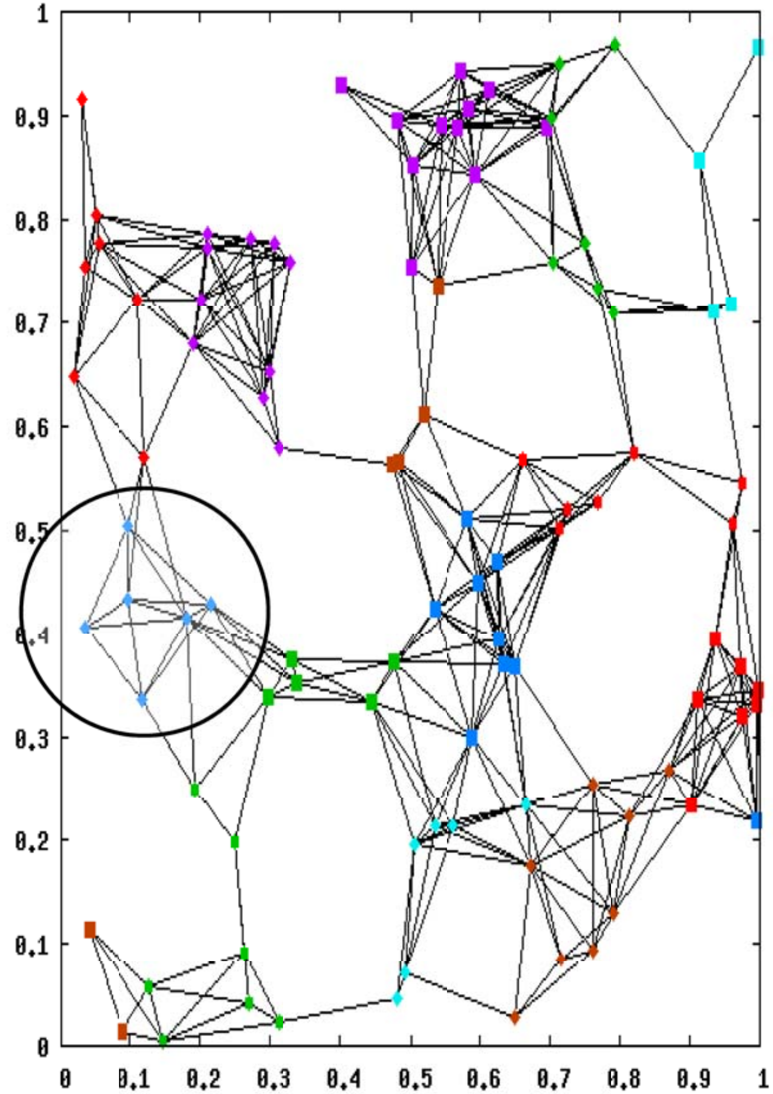
Generation: 56324

Handovers: 19750



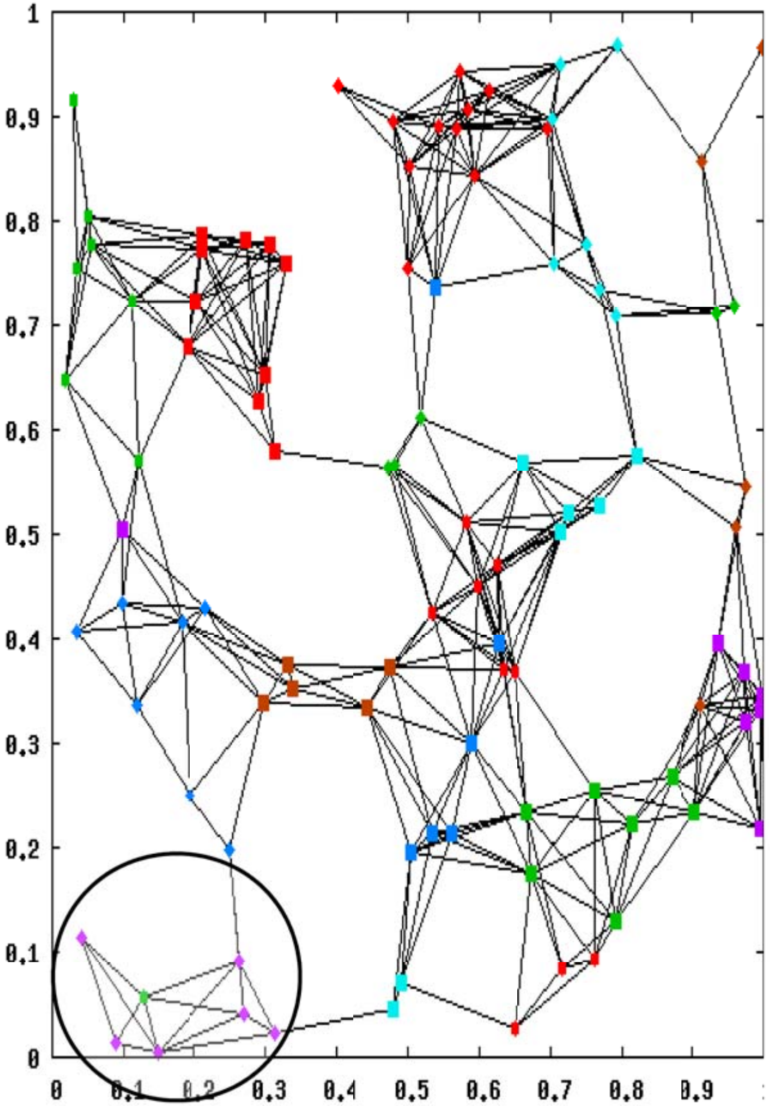
Generation: 1

Handovers: 25872



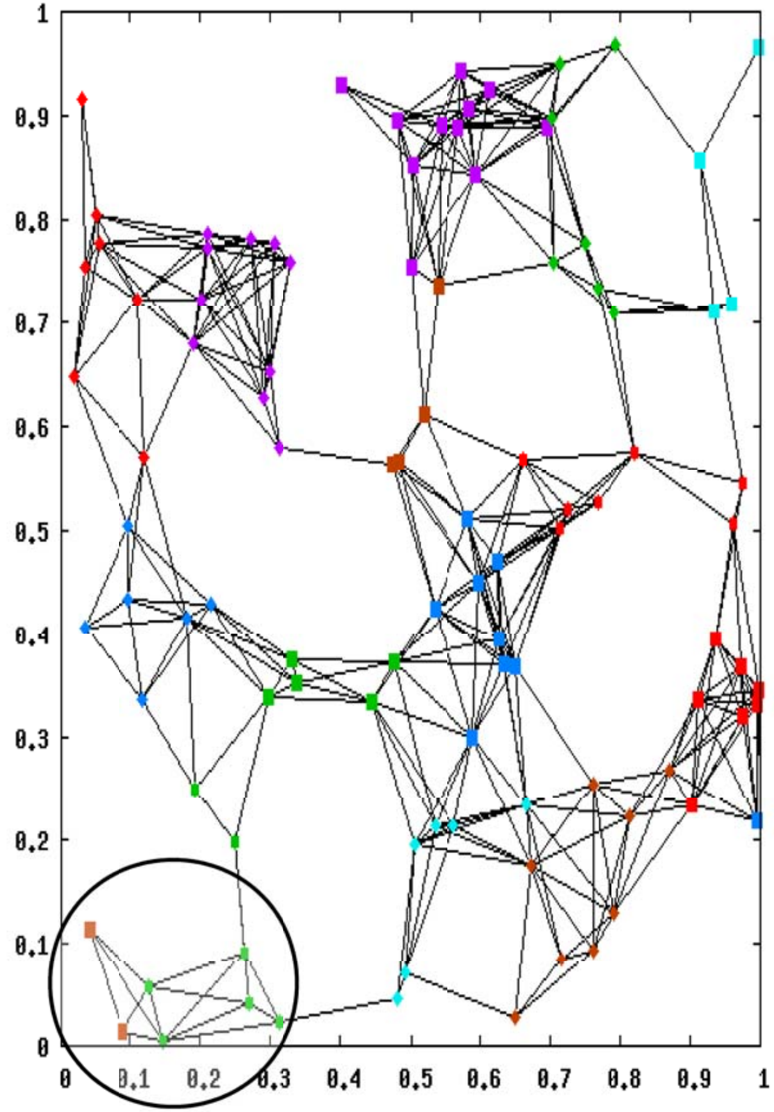
Generation: 56324

Handovers: 19750



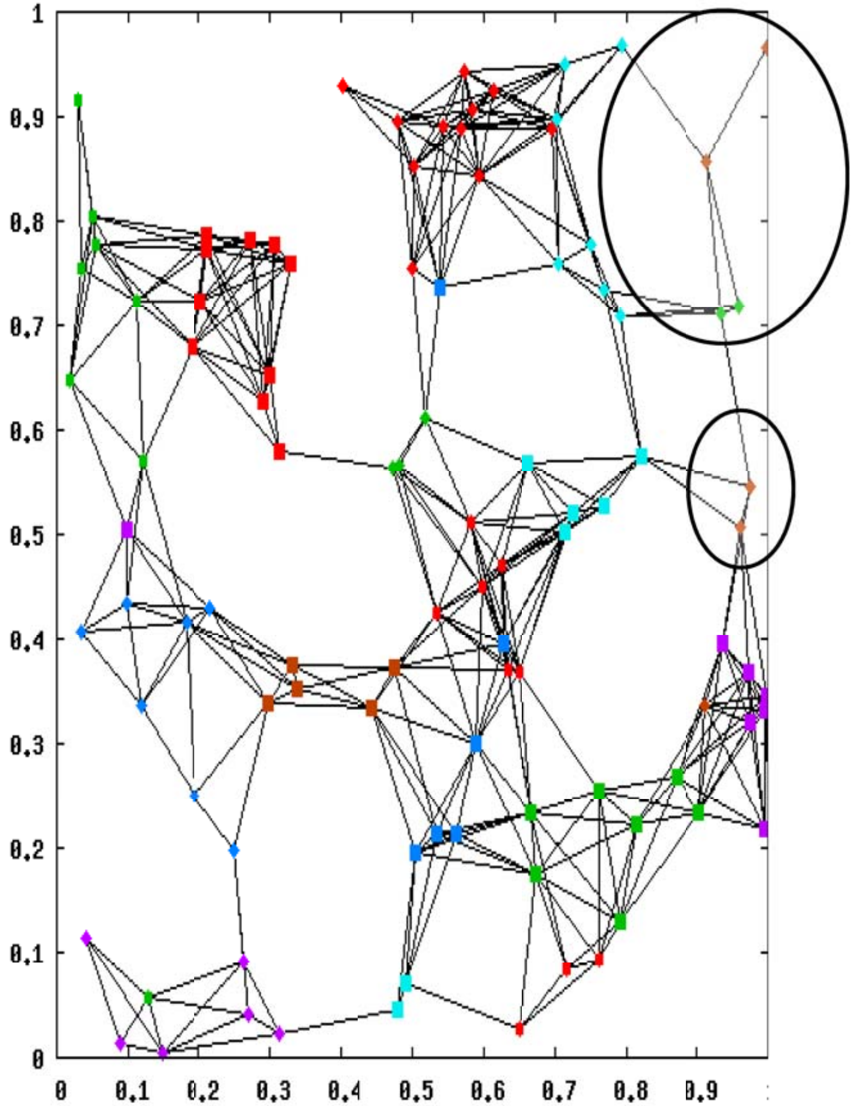
Generation: 1

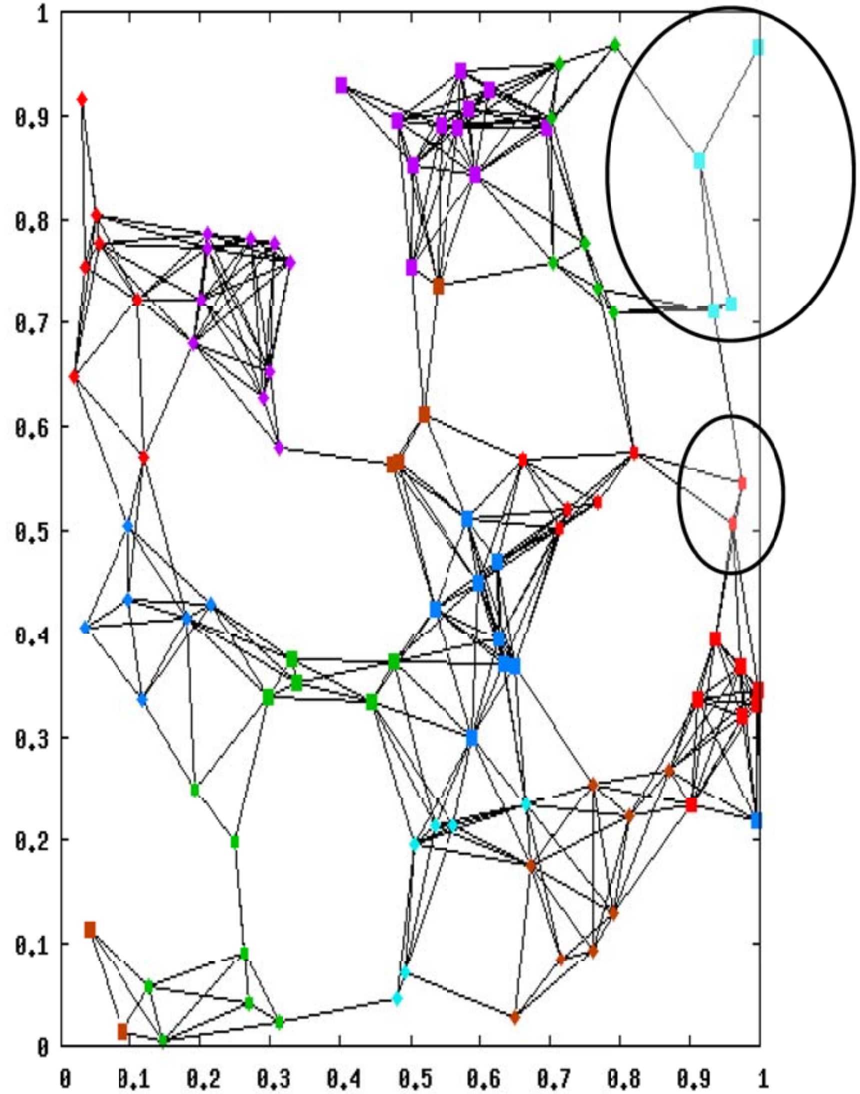
Handovers: 25872

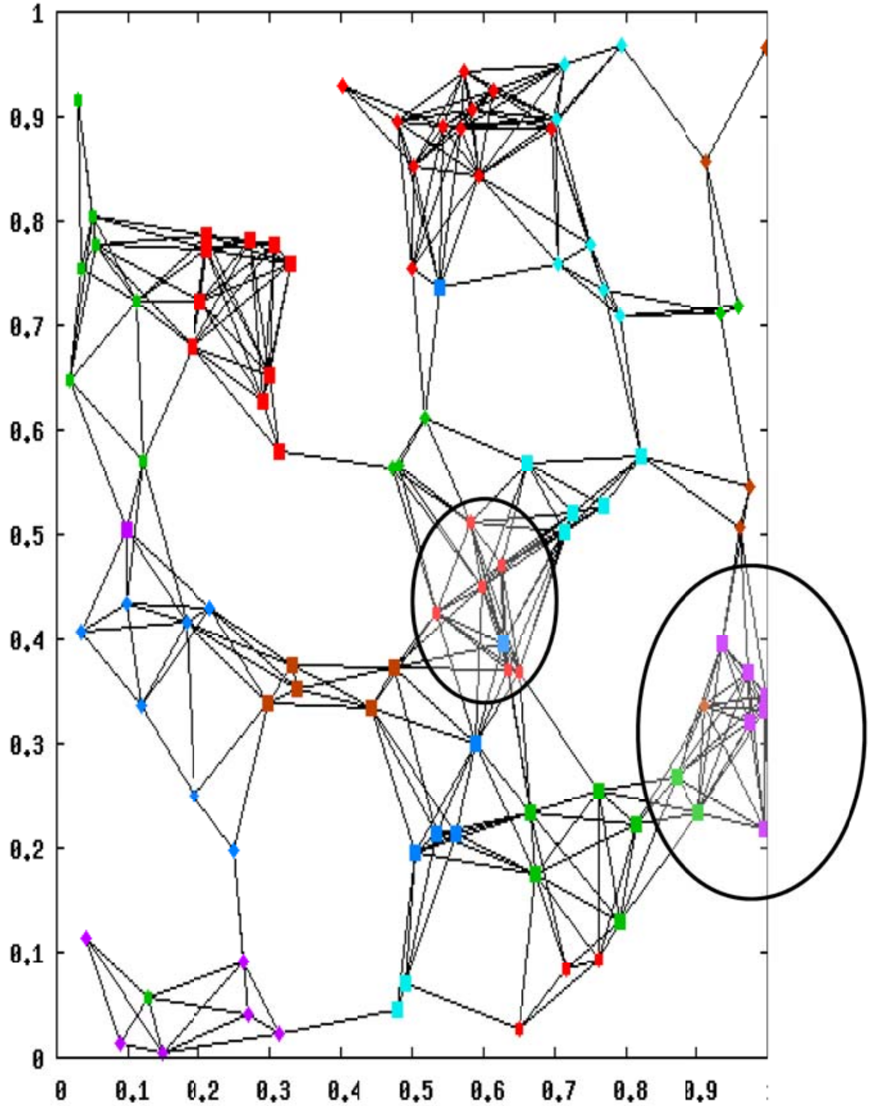


Generation: 56324

Handovers: 19750

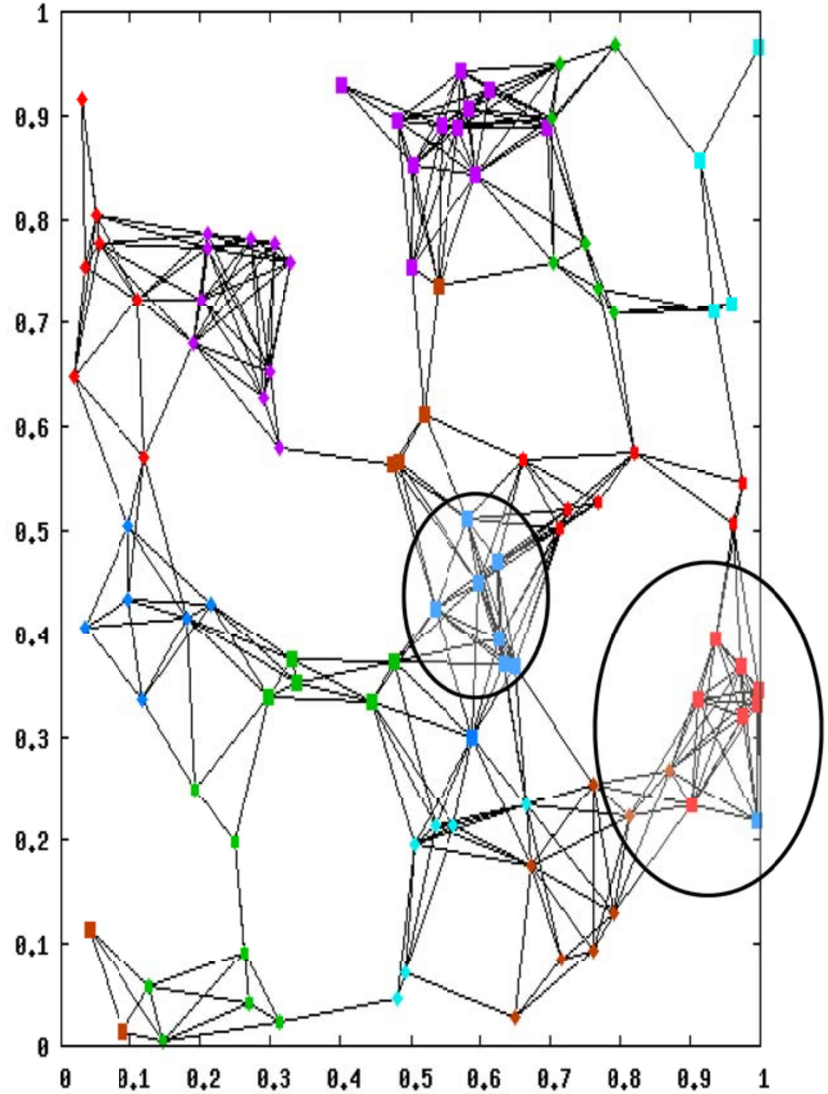

Generation: 1
Handovers: 25872


Generation: 56324
Handovers: 19750



Generation: 1

Handovers: 25872



Generation: 56324

Handovers: 19750

BRKGA for bound constrained global optimization

Bound-constrained global optimization

Find

$$x^* = \text{argmin}\{\ f(x)\ |\ l \leq x \leq u\ \},$$

where $f: R^n \rightarrow R$, and $l, x, u \in R^n$

System of nonlinear equations

Hirsch, Pardalos, M.G.C.R. (2006)

- Given a nonlinear system: $f_1(\mathbf{x}) = 0, \ldots, f_r(\mathbf{x})=0$
- Formulate the optimization problem:

$$\text{Find } \mathbf{x}^* = \text{argmin}\{F(\mathbf{x}) = \textstyle\sum_{i=1\ldots r} f_i^2(\mathbf{x}) \mid l \leq \mathbf{x} \leq u\}$$

- Since $F(\mathbf{x}) \geq 0$ for all $l \leq \mathbf{x} \leq u$, then

$$F(\mathbf{x}) = 0 \Leftrightarrow f_i(\mathbf{x}) = 0 \text{ for all } i \in \{1, \ldots, r\}$$

- Hence if $\exists \; l \leq \mathbf{x}^* \leq u \ni F(\mathbf{x}^*) = 0 \Rightarrow \mathbf{x}^*$ is a global minimizer of problem and $\mathbf{x}^*$ is a root of the system of equations: $f_1(\mathbf{x}) = 0, \ldots, f_r(\mathbf{x})=0$.

System of nonlinear equations

Hirsch, Pardalos, M.G.C.R. (2006)

Suppose the k-th root (roots are denoted $\mathbf{x}_1, \ldots, \mathbf{x}_k$) has been found.

Then solve new problem, with the modified objective function given by:

$$F(\mathbf{x}) = \sum_{i=1..r} f_i^2(\mathbf{x}) + \beta \sum_{j=1..k} e^{-\|\mathbf{x}-\mathbf{x}(j)\|} \chi_\rho\left(\|\mathbf{x}-\mathbf{x}_j\|\right)$$

where

$$\chi_\rho(\delta) = 1 \text{ if } \delta \leq \rho;\ 0, \text{ otherwise}$$

β is a large constant, and ρ is a small constant.

This has the effect of creating an area of repulsion near solutions that have already been found by the heuristic.

BRKGA for bound-constrained global optimization

Encoding & Decoder of BRKGA for global optimization

- A solution is encoded as a vector $\chi = (\chi_1, \ldots, \chi_n)$ of size n, where χ_i is a random number in the interval [0,1], for i=1,...,n. The i-th component of χ corresponds to the i-th dimension of hyper-rectangle S.
- A decoder takes as input the vector of random keys χ and returns a solution $x \in S$ with

$$x_i = l_i + \chi_i \cdot (u_i - l_i), \text{ for } i=1,\ldots,n.$$

During all decoder process, the solutions fitness are calculated by the objective function $f: S \rightarrow R$ of global optimization problem.

Computational environment

Computer with a 1.66GHz Intel Core 2 processor with 1 GB of Memory

Ubuntu version 4.3.2-1ubuntu11

C language, gcc compiler version 4.3.2

Random-number generator: Mersenne Twister algorithm (Matsumoto and Nishimura, 1998)

Robot kinematics problem

Robot kinematics application

- First described by Tsai and Morgan (1985).
- Considered a "challenging problem" in Floudas et al. (1999).
- Given a 6-revolute manipulator (rigid-bodies, or links, connected together by joints), with the first link designated the base, and the last link designated the hand of the robot: Determine the possible positions of the hand, given that the joints are movable.
- Problem is reduced to solving a system of eight nonlinear equations in eight unknowns.

Robot kinematics application

Find $\mathbf{x} = (x_1, x_2, \ldots, x_8)$ such that:

- $f_1(\mathbf{x}) = 4.731 \cdot 10^{-3} x_1 x_3 - 0.3578 x_2 x_3 - 0.1238 x_1 + x_7 - 1.637 \cdot 10^{-3} x_2 - 0.9338 x_4 - 0.3571 = 0$
- $f_2(\mathbf{x}) = 0.2238 x_1 x_3 + 0.7623 x_2 x_3 + 0.2638 x_1 - x_7 - 0.07745 x_2 - 0.6734 x_4 - 0.6022 = 0$
- $f_3(\mathbf{x}) = x_6 x_8 + 0.3578 x_1 + 4.731 \cdot 10^{-3} x_2 = 0$
- $f_4(\mathbf{x}) = -0.7623 x_1 + 0.2238 x_2 + 0.3461 = 0$
- $f_5(\mathbf{x}) = x_1^2 + x_2^2 - 1 = 0$
- $f_6(\mathbf{x}) = x_3^2 + x_4^2 - 1 = 0$
- $f_7(\mathbf{x}) = x_5^2 + x_6^2 - 1 = 0$
- $f_8(\mathbf{x}) = x_7^2 + x_8^2 - 1 = 0$

Parameters of biased random-key GA for robot kinematics application

- Size of chromosome: 8
- Size of population: 10
- Size of elite partition: 20% of population
- Size of mutant set: 10% of population
- Child inheritance probability: 0.7
- Stopping criterion: at any time during a run, we say that the heuristic has solved the problem if $GAP = | F(x) - F(x^*) | \leq \varepsilon$, with $\varepsilon = 0.0001$, where x is the current best solution found by the heuristic and x^* is the known global minimum solution.

Robot kinematics application

- We ran BRKGA five times (a different starting random seed for each run) with $\rho = 1$, $\beta = 10^{10}$
- In each case, BRKGA heuristic was able to find all 16 known roots.
- The average CPU time needed to find the 16 roots was 3623.27 seconds.
- The next table illustrates one of these solutions:

  the 16 roots were found in 4013.27 seconds by running BRKGA heuristic with seed=270001.

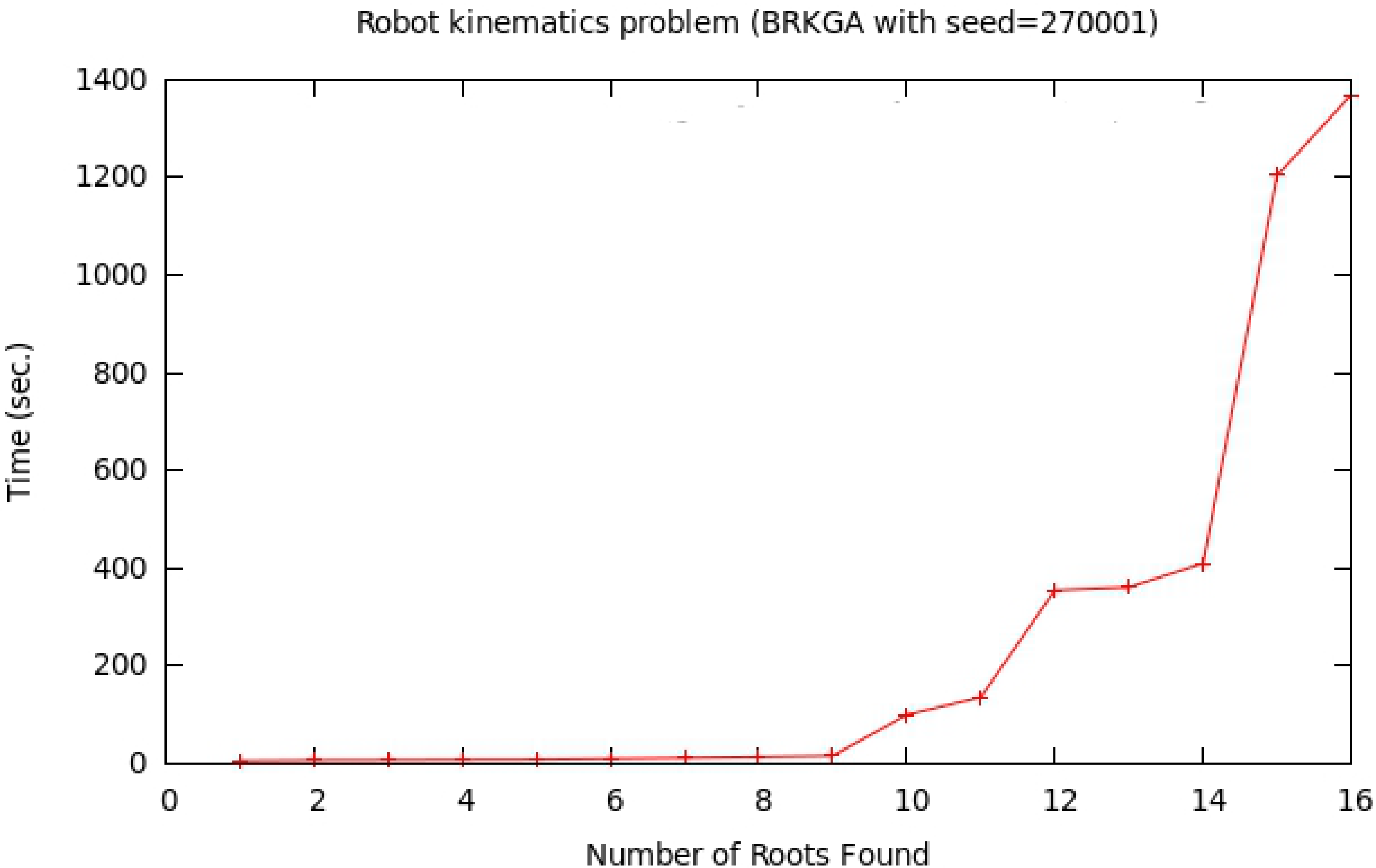
Known roots x=(x1,...,x8) of system in $[-1, 1]^8$ described in Floudas et al. [1999], Kearfott [1987].

x1	x2	x3	x4	x5	x6	x7	x8
0.1644	−0.9864	0.7185	−0.6956	−0.9980	0.0638	−0.5278	−0.8494
0.1644	−0.9864	0.7185	−0.6956	−0.9980	−0.0638	−0.5278	0.8494
0.1644	−0.9864	0.7185	−0.6956	0.9980	−0.0638	−0.5278	0.8494
0.6716	0.7410	−0.6516	−0.7586	−0.9625	−0.2711	−0.4376	0.8992
0.6716	0.7410	−0.6516	−0.7586	0.9625	0.2711	−0.4376	−0.8992
0.6716	0.7410	−0.6516	−0.7586	0.9625	−0.2711	−0.4376	0.8992
0.6716	0.7410	0.9519	−0.3064	−0.9638	0.2666	0.4046	−0.9145
0.6716	0.7410	0.9519	−0.3064	0.9638	−0.2666	0.4046	0.9145
0.6716	0.7410	−0.6516	−0.7586	−0.9625	0.2711	−0.4376	−0.8992
0.6716	0.7410	0.9519	−0.3064	0.9638	0.2666	0.4046	−0.9145
0.6716	0.7410	0.9519	−0.3064	−0.9638	−0.2666	0.4046	0.9145
0.1644	−0.9864	−0.9471	−0.3210	−0.9982	−0.0594	0.4110	0.9116
0.1644	−0.9864	−0.9471	−0.3210	0.9982	−0.0594	0.4110	0.9116
0.1644	−0.9864	−0.9471	−0.3210	−0.9982	0.0594	0.4110	−0.9116
0.1644	−0.9864	−0.9471	−0.3210	0.9982	0.0594	0.4110	−0.9116
0.1644	−0.9864	0.7185	−0.6956	0.9980	0.0638	−0.5278	−0.8494

Known roots x=(x1,...,x8) of system in $[-1, 1]^8$ described in Floudas et al. [1999], Kearfott [1987].

Roots x=(x1,...,x8) of system in $[-1, 1]^8$ found by running BRKGA with seed=270001. For each root, the time (seconds) and the value of obj. function F(x) are shown in the first column.

	x1	x2	x3	x4	x5	x6	x7	x8
4.95 s	0.1658	−0.9851	0.7153	−0.6950	−0.9975	0.0638	−0.5251	−0.8557
$9.79321\ 10^{-5}$	0.1644	−0.9864	0.7185	−0.6956	−0.9980	0.0638	−0.5278	−0.8494
7.5 s	0.1619	−0.9851	0.7182	−0.6946	−0.9979	−0.0616	−0.5232	0.8503
$7.19678\ 10^{-5}$	0.1644	−0.9864	0.7185	−0.6956	−0.9980	−0.0638	−0.5278	0.8494
13.19 s	0.1731	−0.9827	0.7181	−0.6946	0.9973	−0.0686	−0.5195	0.8544
$9.54526\ 10^{-5}$	0.1644	−0.9864	0.7185	−0.6956	0.9980	−0.0638	−0.5278	0.8494
5.95 s	0.6729	0.7394	−0.6480	−0.7641	−0.9623	−0.2706	−0.4333	0.9027
$9.76283\ 10^{-5}$	0.6716	0.7410	−0.6516	−0.7586	−0.9625	−0.2711	−0.4376	0.8992
6.86 s	0.6736	0.7383	−0.6505	−0.7553	0.9634	0.2696	−0.4333	−0.9010
$6.49664\ 10^{-5}$	0.6716	0.7410	−0.6516	−0.7586	0.9625	0.2711	−0.4376	−0.8992
6.53 s	0.6792	0.7328	−0.6555	−0.7553	0.9612	−0.27613	−0.4343	0.9027
$9.23596\ 10^{-5}$	0.6716	0.7410	−0.6516	−0.7586	0.9625	−0.2711	−0.4376	0.8992
11.05 s	0.6768	0.7358	0.9502	−0.3132	−0.9623	0.2708	0.4002	−0.9162
$9.68334\ 10^{-5}$	0.6716	0.7410	0.9519	−0.3064	−0.9638	0.2666	0.4046	−0.9145
15.24 s	0.6674	0.7427	0.9508	−0.3132	0.9661	−0.2620	0.4002	0.9156
$9.81702\ 10^{-5}$	0.6716	0.7410	0.9519	−0.3064	0.9638	−0.2666	0.4046	0.9145
9.16 s	0.6792	0.7362	−0.6564	−0.7553	−0.9617	0.2745	−0.4343	−0.9008
$9.1171\ 10^{-5}$	0.6716	0.7410	−0.6516	−0.7586	−0.9625	0.2711	−0.4376	−0.8992
98.98 s	0.6707	0.7462	0.953	−0.3041	0.9644	0.2631	0.4079	−0.9107
$8.55693\ 10^{-5}$	0.6716	0.7410	0.9519	−0.3064	0.9638	0.2666	0.4046	−0.9145
135.02 s	0.6646	0.749	0.9551	−0.3015	−0.9652	−0.2625	0.4101	0.9114
$9.82556\ 10^{-5}$	0.6716	0.7410	0.9519	−0.3064	−0.9638	−0.2666	0.4046	0.9145
354.76 s	0.1604	−0.9891	−0.9505	−0.3167	−0.9979	−0.0581	0.4111	0.909
$9.32723\ 10^{-5}$	0.1644	−0.9864	−0.9471	−0.3210	−0.9982	−0.0594	0.4110	0.9116
360.76 s	0.168	−0.9844	−0.9514	−0.3167	0.9998	−0.0602	0.4098	0.9124
$9.70348\ 10^{-5}$	0.1644	−0.9864	−0.9471	−0.3210	0.9982	−0.0594	0.4110	0.9116
409.27 s	0.1606	−0.9855	−0.9481	−0.3183	−0.9976	0.0554	0.4138	−0.9076
$7.28536\ 10^{-5}$	0.1644	−0.9864	−0.9471	−0.3210	−0.9982	0.0594	0.4110	−0.9116
1204.24 s	0.1712	−0.9850	−0.9427	−0.3275	0.9976	0.0621	0.4052	−0.9143
$8.21721\ 10^{-5}$	0.1644	−0.9864	−0.9471	−0.3210	0.9982	0.0594	0.4110	−0.9116
1369.81 s	0.1718	−0.9837	0.7178	−0.6947	0.9943	0.0687	−0.5246	−0.8519
$8.63659\ 10^{-5}$	0.1644	−0.9864	0.7185	−0.6956	0.9980	0.0638	−0.5278	−0.8494


Robot kinematics problem (BRKGA with seed=270001)
Time (sec.)
Number of Roots Found
0
200
400
600
800
1000
1200
1400
0
2
4
6
8
10
12
14
16

Chemical reaction engineering

Non-Isothermal CSTR (continuously stirred tank reactors) problem

Originally described in Kubicek et al. (1980)

This problem concerns a model of two continuous non-adiabatic stirred tank reactors. These reactors are in series, in steady state, with a recycle component, and have an exothermic first-order irreversible reaction.

When certain variables are eliminated, the model results in a system of two nonlinear equations ...

Non-Isothermal CSTR (continuously stirred tank reactors) problem

$$f_1 = (1-R)\left[\frac{D}{10(1+\beta_1)} - \phi_1\right]\exp\left(\frac{10\phi_1}{1+10\phi_1/\gamma}\right) - \phi_1$$

$$f_2 = \phi_1 - (1+\beta_2)\phi_2 + (1-R)\times$$

$$[D/10 - \beta_1\phi_1 - (1+\beta_2)\phi_2]\exp\left(\frac{10\phi_2}{1+10\phi_2/\gamma}\right)$$

$\phi 1$ and $\phi 2$ represent the dimensionless temperatures in the two reactors in the domain $[0,1]^2$.

Parameters γ, D, $\beta 1$, and $\beta 1$ are set to 1000, 22, 2, and 2, respectively. The recycle ratio parameter R takes on values in the set $\Re$={0.935, 0.940, ... , 0.995}, whose number of known solutions varies between 1 and 7.

Parameters of BRKGA for Non-Isothermal CSTR problem

- Size of chromosome: 8
- Size of population: 100
- Size of elite partition: 20% of population
- Size of mutant set: 10% of population
- Child inheritance probability: 0.7
- Stopping criterion: at any time during a run, we say that the heuristic has solved the problem if $GAP = |\ F(x) - F(x^*)\ | \leq \varepsilon$, with $\varepsilon = 0.000001$, where x is the current best solution found by the heuristic and x^* is the known global minimum solution.

For each value of the parameter R given in the set $\Re$, we ran the BRKGA heuristic 5 times, each time searching for all of roots.

R	#sols.	C-GRASP avg.#found	C-GRASP avg. time	BRKGA avg.#found	BRKGA avg. time
0.935	1	1.00	0.60s	1.00	0.822s
0.940	1	1.00	0.77s	1.00	0.635s
0.945	3	3.00	0.19s	3.00	0.876s
0.950	5	4.99	1.11s	4.65	1.760s
0.955	5	5.00	1.69s	5.00	2.342s
0.960	7	6.96	2.41s	6.87	2.375s
0.965	5	4.95	1.81s	4.78	2.054s
0.970	5	4.99	1.34s	4.82	1.732s
0.975	5	4.96	1.83s	4.76	2.012s
0.980	5	4.98	1.90s	4.92	2.759s
0.985	5	4.99	2.23s	4.95	4.310s
0.990	1	1.00	0.01s	1.00	0.018s
0.995	1	1.00	0.01s	1.00	0.034s

Automotive engineering problem

Automotive steering problem

- Kinematic synthesis mechanism for automotive steering.
- This problem was originally described in Pramanik (2002).
- The Ackerman steering mechanism is a four-bar mechanism for steering four-wheel vehicles. When a vehicle turns, the steered wheels need to be angled so that they are both 90° with respect to a certain line. This means that the wheels will have to be at different angles with respect to the non-steered wheels. The Ackerman design arranges the wheels automatically by moving the steering pivot inward.
- Pramanik states that "the Ackerman design reveals progressive deviations from ideal steering with increasing ranges of motion."
- Pramanik instead considers a six-member mechanism. This produces the system of equations given ...

$$G_i(\psi_i, \phi_i) = \Big[E_i(x_2 \sin(\psi_i) - x_3) - F_i(x_2 \sin(\phi_i) - x_3)\Big]^2 +$$

i = 0, 1, 2, 3

$$\Big[F_i(1 + x_2 \cos(\phi_i)) - E_i(x_2 \cos(\psi_i) - 1)\Big]^2 -$$

$$\Big[(1 + x_2 \cos(\phi_i))(x_2 \sin(\psi_i) - x_3)x_1 -$$

$$(x_2 \sin(\phi_i) - x_3)(x_2 \cos(\psi_i) - x_3)x_1\Big]^2 = 0$$

where

$$E_i = x_2(\cos(\phi_i) - \cos(\phi_0)) - x_2 x_3(\sin(\phi_i) - \sin(\phi_0)) - (x_2 \sin(\phi_i) - x_3)x_1$$

and

$$F_i = -x_2 \cos(\psi_i) - x_2 x_3 \sin(\psi_i) + x_2 \cos(\psi_0) + x_1 x_3 + (x_3 - x_1)x_2 \sin(\psi_0).$$

We want to find x_1, x_2, x_3 such that

$F(x) = G_0(\Psi_0,\phi_0)^2 + G_1(\Psi_1,\phi_1)^2 + G_2(\Psi_2,\phi_2)^2 + G_3(\Psi_3,\phi_3)^2$ is minimized, where

x_1, x_2, and x_3 are, respectively, the normalized steering pivot rod radius, the normalized tire pivot radius, and the normalized 'length' direction distance from the steering rod pivot point to the tire pivot.

Parameters of biased random-key GA for automotive steering problem

- Size of chromosome: 8
- Size of population: 100
- Size of elite partition: 20% of population
- Size of mutant set: 10% of population
- Child inheritance probability: 0.7
- Stopping criterion: at any time during a run, we say that the heuristic has solved the problem if $GAP = | F(x) - F(x^*) | \leq \varepsilon$, with $\varepsilon = 0.000001$, where x is the current best solution found by the heuristic and x^* is the known global minimum solution. Here, we know $F(x^*) = 0$.

When the angles ψ_i and ϕ_i are given as:

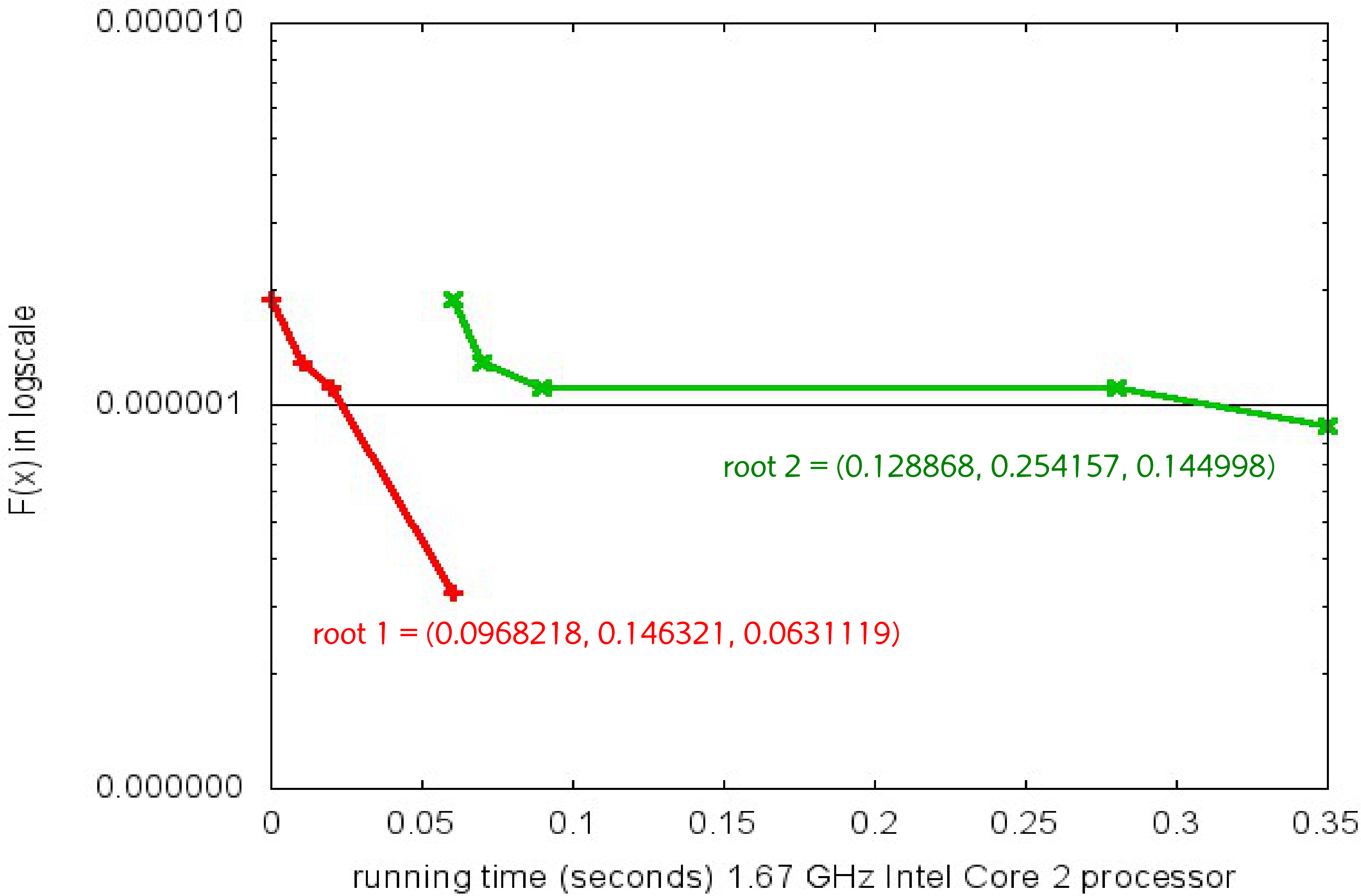
i	ψ_i	ϕ_i
0	1.3954170041747090114	1.7461756494150842271
1	1.7444828545735749268	2.0364691127919609051
2	2.0656234369405315689	2.2390977868265978920
3	2.4600678478912500533	2.4600678409809344550

This system had two roots in the domain $[0.06, 1]^3$.

Using BRKGA, we solved the problem 10 times.

Each time, BRKGA found the two roots of the system .

Computing two roots of system



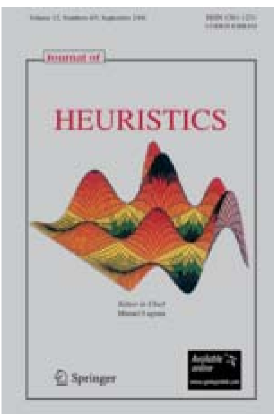
Literature survey

Literature

- BRKGAs have been applied in a wide range of areas.
- The following is a sampling of some papers that appeared in the literature applying BRKGAs.

Survey

- Survey: Gonçalves and R. (2011)



J.F. Gonçalves and M.G.C.R., "Biased random-key genetic algorithms for combinatorial optimization," J. of Heuristics, vol.17, pp. 487-525, 2011.

Telecommunications

- Routing: Ericsson, R., Pardalos (2002), Buriol et al. (2002, 2005), Reis et al. (2011), Noronha, R., Ribeiro (2007, 2008, 2011), Heckeler et al. (2011)
- Design: Andrade et al. (2006), Buriol, R., Thorup (2007)
- Network monitoring: Breslau et al. (2011)
- Regenerator location: Duarte et al. (2011)
- Fiber installation in optical networks: Goulart et al. (2011)
- Path-based recovery in flexgrid optical networks: Castro et al. (2012)

Telecommunications (cont'd)

- Handover minimization: Morán-Mirabal et al. (2012)
- Survivable IP/MPLS-over-WSON multi-layer network: Ruiz et al. (2011), Pedrola et al. (2011)
- Survey: R. (2012)

Scheduling

- Job-shop scheduling: Gonçalves, Mendes, R. (2005), Gonçalves and R. (2012)
- Single machine scheduling: Valente et al. (2006), Valente and Gonçalves (2008)
- Resource constrained project scheduling: Gonçalves, Mendes, R. (2008, 2009), Gonçalves, R., Mendes (2011)
- Selection and scheduling of observations on Earth observing satellites: Tangpattanakul, Josefowiez, Lopez (2012)

Production planning

- Assembly line balancing: Gonçalves and Almeida (2002)
- Manufacturing cell formation: Gonçalves and R. (2004)
- Single machine scheduling: Valente et al. (2006), Valente and Gonçalves (2008)
- Assembly line worker assignment and balancing: Moreira et al. (2010)
- Lot sizing and scheduling with capacity constraints and backorders: Gonçalves and Sousa (2011)

Network optimization

- Concave minimum cost flow: Fontes and Gonçalves (2007)
- Robust shortest path: Coco, Noronha, Santos (2012)
- Tree of hubs location: Pessoa, Santos, R. (2012)
- Hop-constrained trees in nonlinear cost flow networks: Fontes and Gonçalves (2012)
- Capacitated arc routing: Martinez, Loiseau, R. (2011)

Power systems

- Unit commitment: Roque, Fontes, Fontes (2010, 2011)
- Multi-objective unit commitment: Roque, Fontes, Fontes (2012)

Packing

- 2D orthogonal packing: Gonçalves and R. (2011)
- 3D container loading: Gonçalves and R. (2012a)
- 2D/3D bin packing: Gonçalves and R. (2012b)

Covering

- Steiner triple systems: R. et al. (2012)
- Covering by pairs: Breslau et al. (2011)

Transportation

- Tollbooth assignment: Buriol. et al. (2009, 2010)

Auctions

- Combinatorial auctions: Andrade et al. (2012)

Automatic parameter tuning

- GRASP with path-relinking: Festa et al. (2010)
- GRASP with evolutionary path-relinking: Morán-Mirabal, González-Velarde, R. (2012)

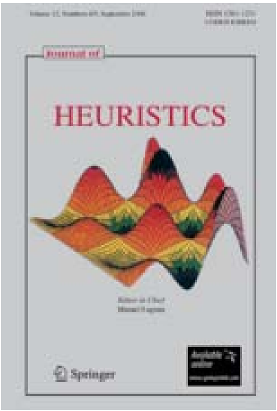
Continuous global optimization

- Bound-constrained optimization: Silva, Pardalos, R. (2012)

Software

- C++ API: Toso and R. (2012)

Reference



J.F. Gonçalves and M.G.C.R., "Biased random-key genetic algorithms for combinatorial optimization," J. of Heuristics, vol.17, pp. 487-525, 2011.

Tech report version:

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

Thanks!

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

http://www.research.att.com/~mgcr