Random-Key Optimizers

A tutorial

Talk given at 2024 Symposium of the Brazilian
Operational Research Society (LVI SBPO)
Fortaleza, Brazil ~ November 4-7, 2024

Team Members

Antonio A. Chaves
Univ. Fed. of São Paulo

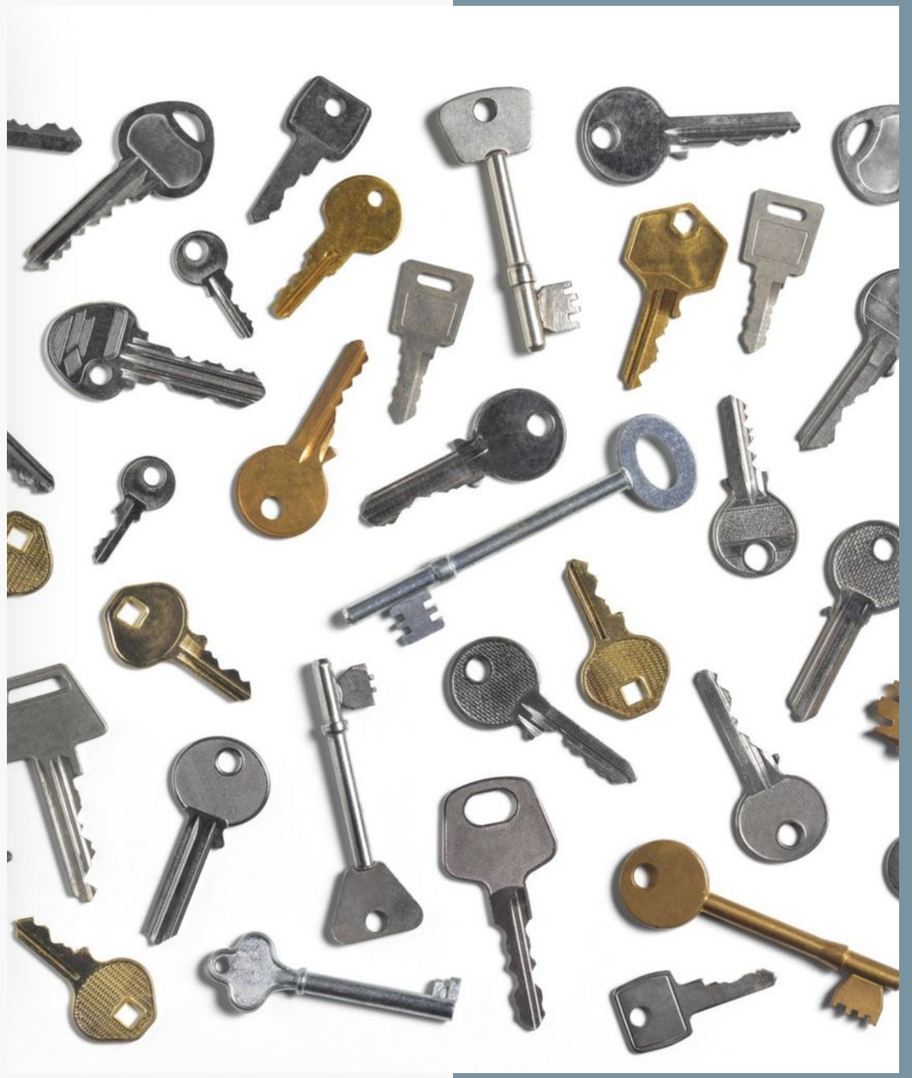
Mauricio G. C. Resende
U. of Washington &
Univ. Fed. of São Paulo

Ricardo M. A. Silva
Univ. Fed. of
Pernambuco

Part I

Contents:

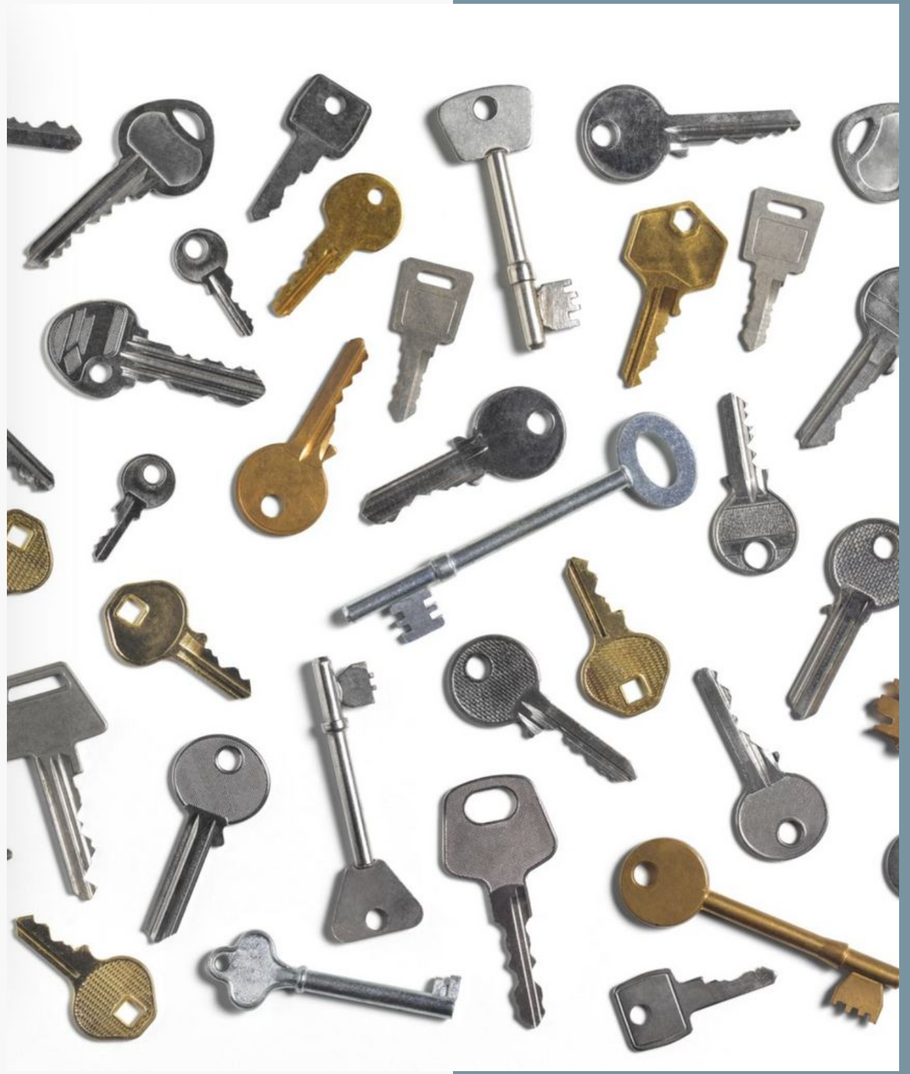
- Encoding and decoding with vectors of random keys
- Random-Key Optimizers (RKO)
  - Components
  - Local Search
  - Metaheuristics



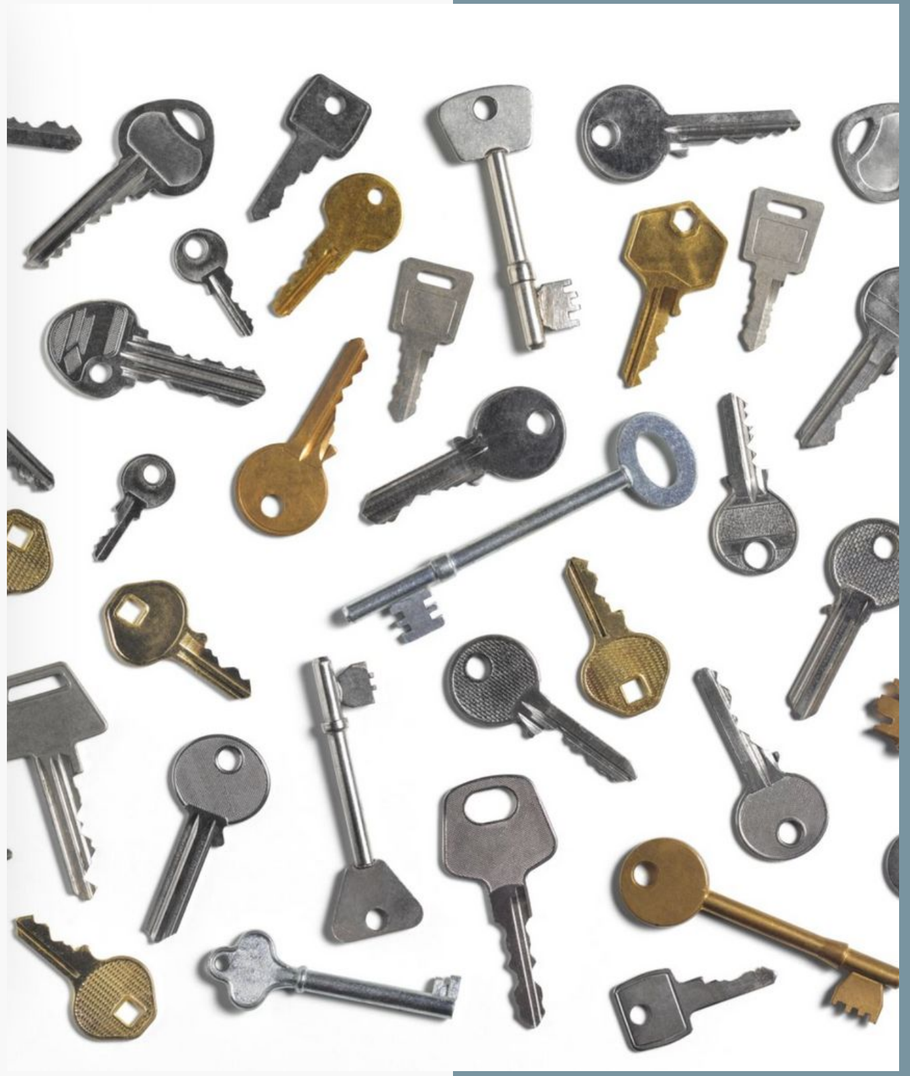
Part II

Contents:

- Examples of encoding and decoding with random keys
- RKO applications in discrete optimization problems
- API



Encoding and decoding with random keys



Encoding and decoding with random keys

Encoding

- Solutions to combinatorial optimization problems can be encoded with an *n*-vector of random keys
- A random key is a randomly generated real number in the interval [0,1)
- An encoded solution is a point in the half-open real unit hypercube of dimension *n*

Bean (1994)

Decoding

- A decoder is a deterministic algorithm (usually a heuristic)
  - INPUT: *n*-vector of random keys
  - OUTPUT: a feasible solution of the combinatorial optimization problem and its cost

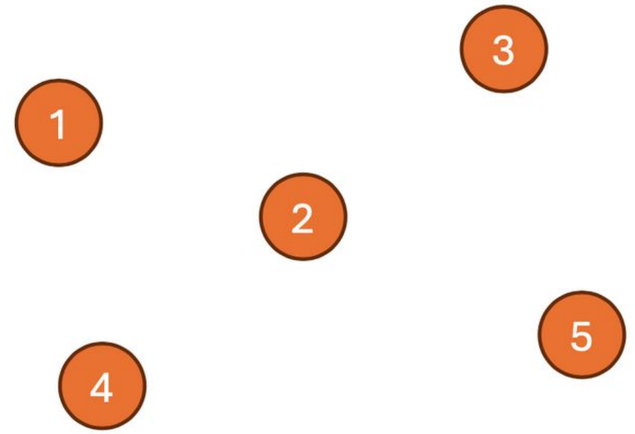
Encoding and decoding: TSP on n cities

Encoding

- Solutions are encoded with an *n*-vector X of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order
- OUTPUT: indices of sorted vector is a permutation of cities visited in tour and total length L of tour



X = (0.085, 0.277, 0.149, 0.332, 0.148)
s[X] = (0.085, 0.148, 0.149, 0.277, 0.332)
π(s[X]) = (1, 5, 3, 2, 4)

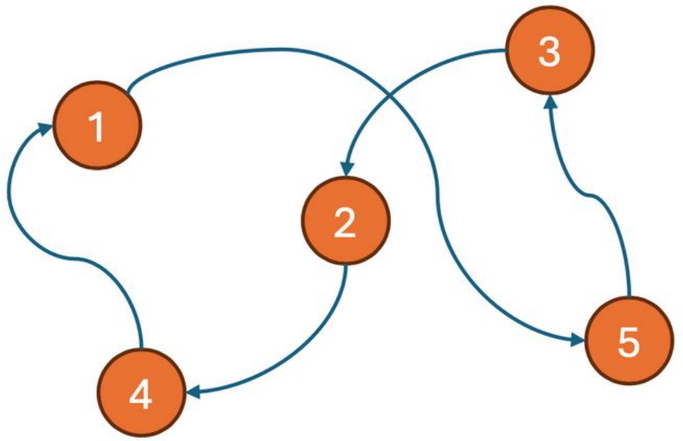
Encoding and decoding: TSP on n cities

Encoding

- Solutions are encoded with an *n*-vector X of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order
- OUTPUT: indices of sorted vector is a permutation of cities visited in tour and total length L of tour



X = (0.085, 0.277, 0.149, 0.332, 0.148)
s[X] = (0.085, 0.148, 0.149, 0.277, 0.332)
π(s[X]) = (1, 5, 3, 2, 4)
L = l(1,5) + l(5,3) + l(3,2) + l(2,4) + l(4,1)

Capacitated vehicle routing with time windows

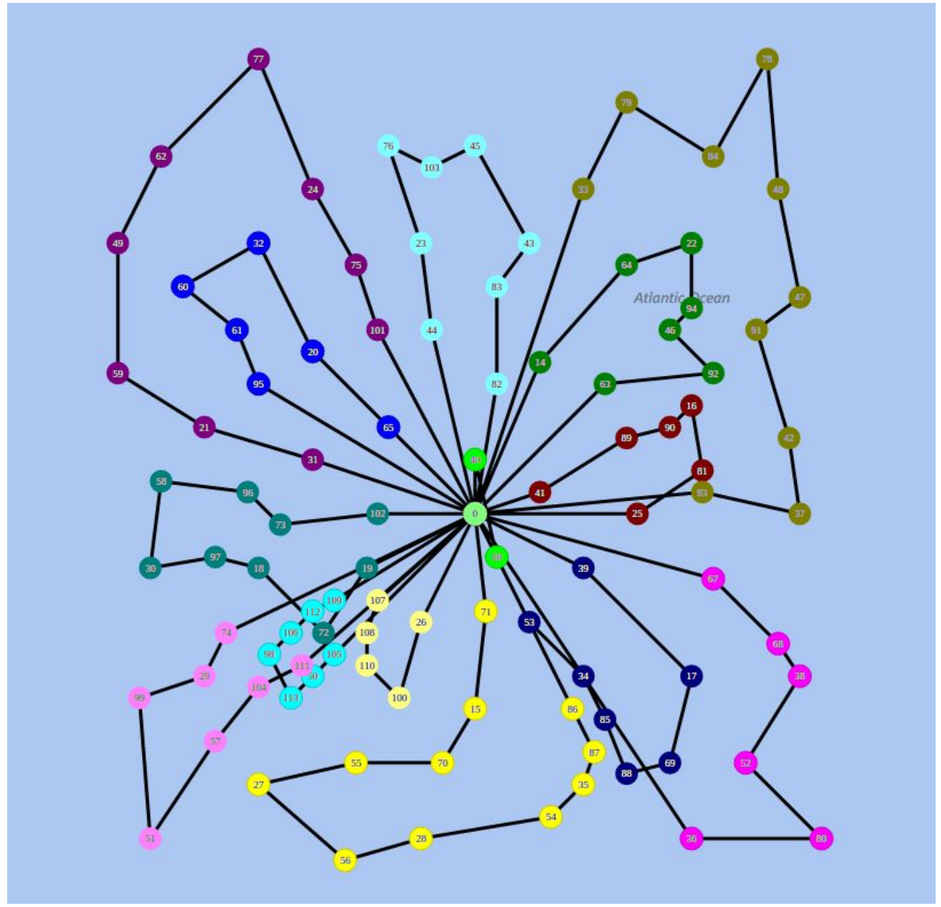
(Resende and Werneck, 2015)

Encoding

- Solutions are encoded with an *n*-vector *X* of N+M random keys

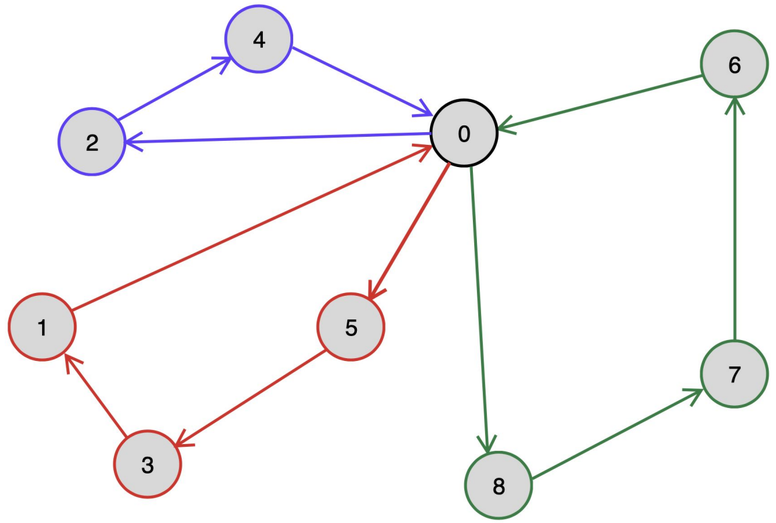
Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order: The last M keys serve as the demarcation of customers assigned to the vehicles. Sorted keys can be rotated so that the index of last key in the vector is always a vehicle key.
- OUTPUT: M routes



Capacitated vehicle routing with time windows

(Resende and Werneck, 2015)



Use penalties to deal with infeasibilities

Encoding and decoding: Set covering with n elements

(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|

1									
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	
0	0	1	0	0	0	1	1	0	
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, 5, 3, 6, 2, 8, 4, 9, 7)
|S| = 1

Encoding and decoding: Set covering with n elements

(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|

1				1					
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	
0	0	1	0	0	0	1	1	0	
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, 3, 6, 2, 8, 4, 9, 7)
|S| = 2

Encoding and decoding: Set covering with n elements

(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|

1		1		1					
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, 6, 2, 8, 4, 9, 7)
|S| = 3

Encoding and decoding: Set covering with n elements

(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|

1		1		1	1				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, 2, 8, 4, 9, 7)
|S| = 4

Encoding and decoding: Set covering with n elements

(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|

1	1	1		1	1				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	◇
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 5

Encoding and decoding: Set covering with n elements

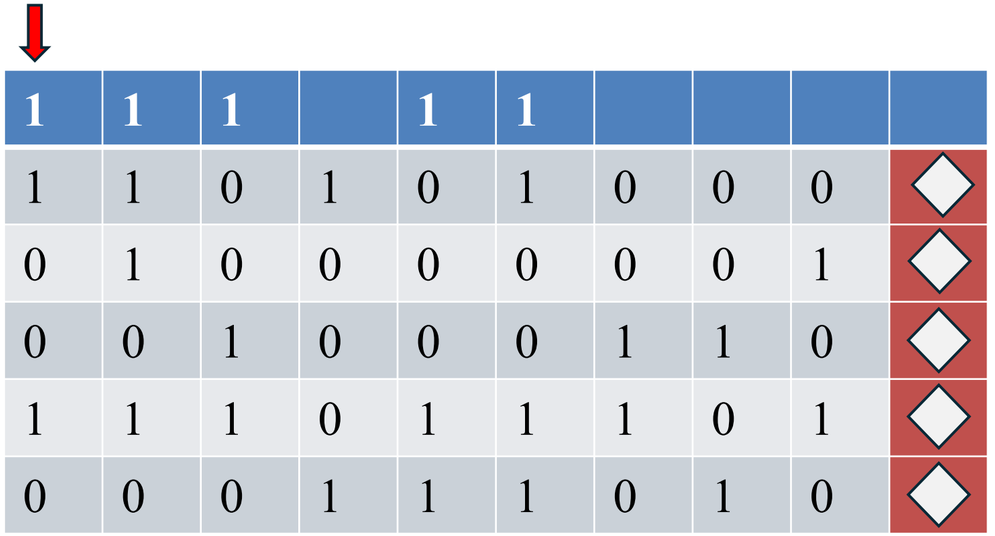
(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|



X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 5

Encoding and decoding: Set covering with n elements

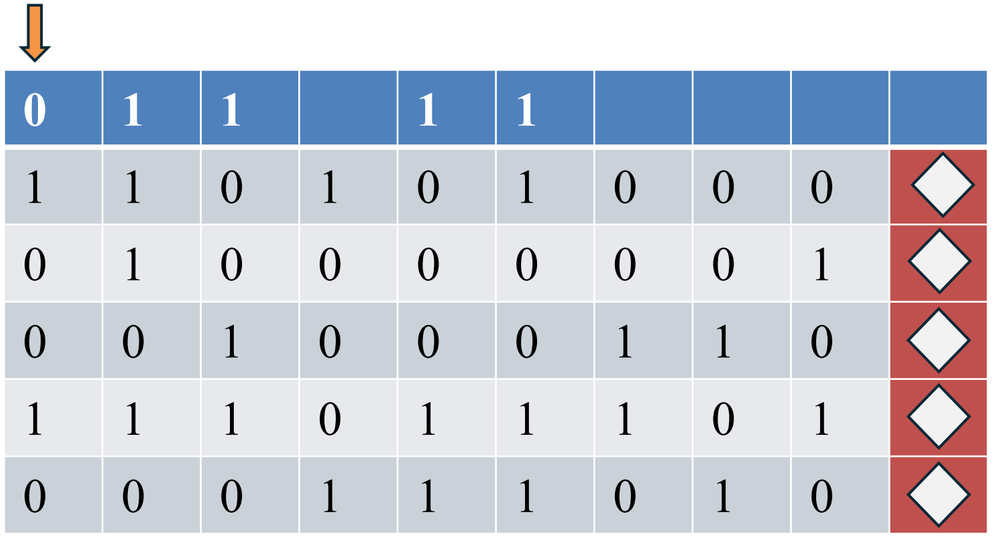
(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|



0	1	1		1	1				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	◇
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 4

Encoding and decoding: Set covering with n elements

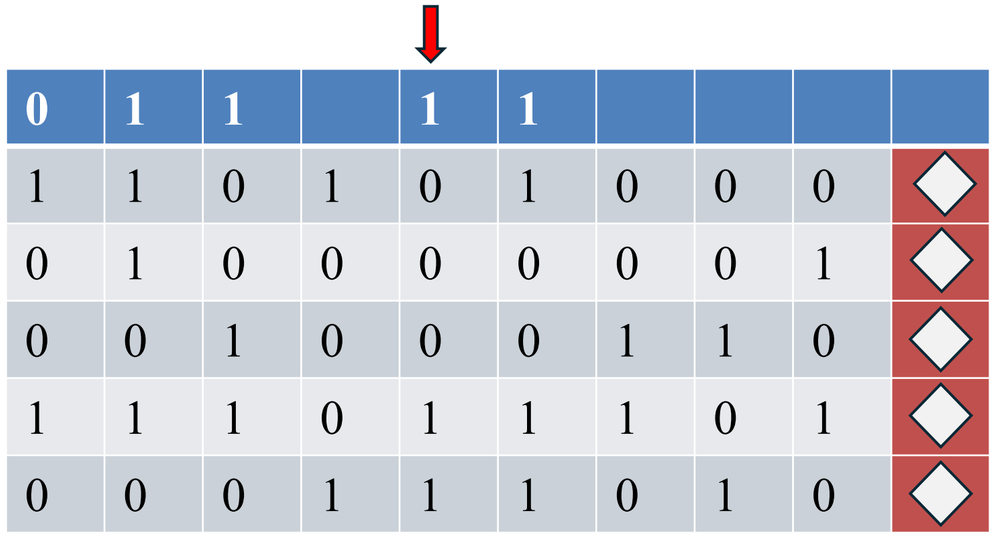
(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|



0	1	1		1	1				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	◇
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 4

Encoding and decoding: Set covering with n elements

(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|

0	1	1		0	1				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	◇
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 3

Encoding and decoding: Set covering with n elements

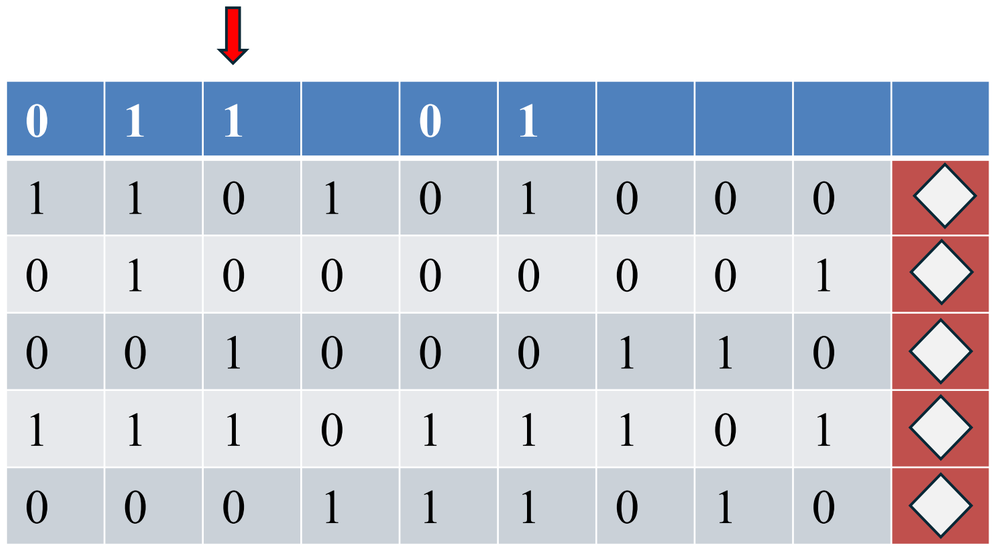
(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|



0	1	1		0	1				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	◇
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 3

Encoding and decoding: Set covering with n elements

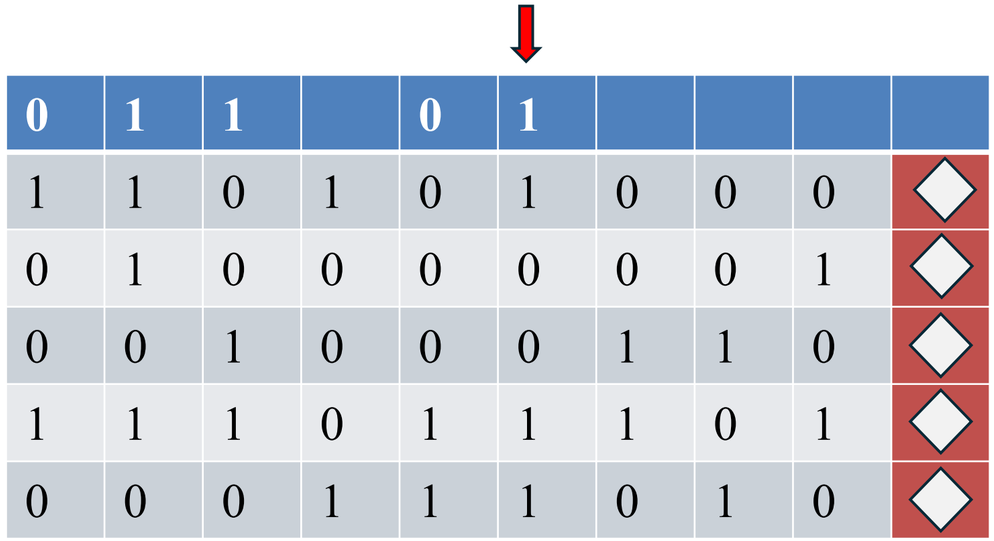
(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|



0	1	1		0	1				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	◇
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 3

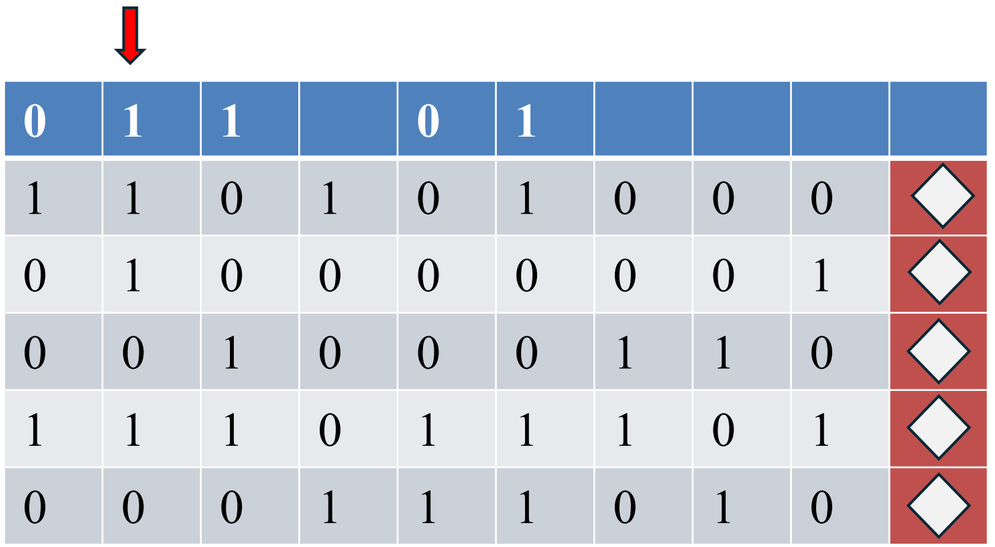
Encoding and decoding: Set covering with n elements
(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|



0	1	1		0	1				
1	1	0	1	0	1	0	0	0	
0	1	0	0	0	0	0	0	1	
0	0	1	0	0	0	1	1	0	
1	1	1	0	1	1	1	0	1	
0	0	0	1	1	1	0	1	0	

X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 3

Encoding and decoding: Set covering with n elements

(Resende et al., 2012)

Encoding

- Solutions are encoded with an *n*-vector *X* of random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the *n* keys in increasing order. : Indices of sorted vector are sequence that elements are placed one at a time in the partial cover. Scan again removing superfluous elements.
- OUTPUT: indices S of selected elements in and total cover size |*S*|

1	2	3	4	5	6	7	8	9	
0	**1**	**1**		**0**	**1**				
1	1	0	1	0	1	0	0	0	◇
0	1	0	0	0	0	0	0	1	◇
0	0	1	0	0	0	1	1	0	◇
1	1	1	0	1	1	1	0	1	◇
0	0	0	1	1	1	0	1	0	◇

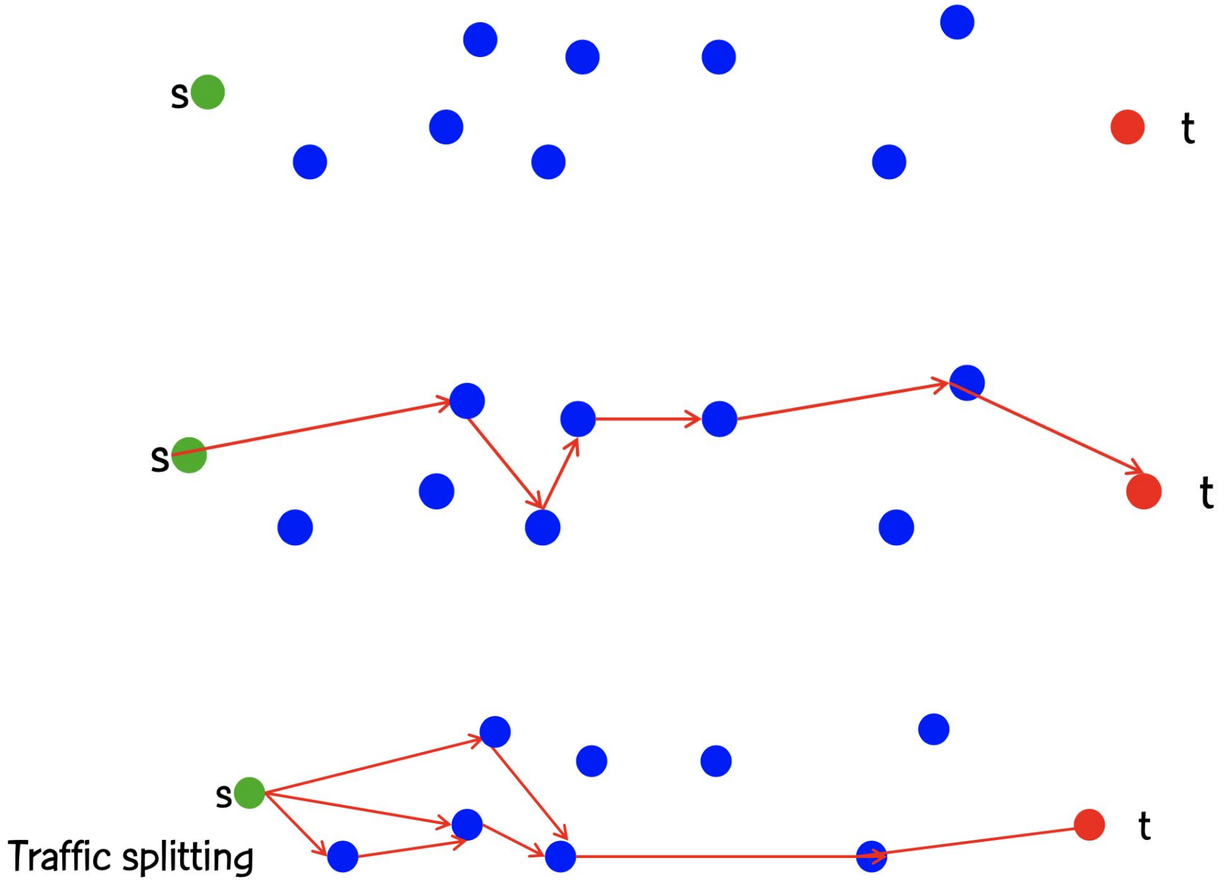
X = (.08, .28, .15, .33, .14, .16, .84, .31, .41)
s[X] = (.08, .14, .15, .16, .28, .31, .33, .41, .84)
π(s[X]) = (**1**, **5**, **3**, **6**, **2**, 8, 4, 9, 7)
|S| = 3

S = {2, 3, 6}

Encoding and decoding: OSPF routing in traffic engineering

(Ericsson et al., 2002; Buriol et al., 2005)

- Given a network G = (N, E), where N is the set of routers and E is the set of links.
- The OSPF (**open shortest path first**) routing protocol assumes each link *e* has a weight *w(e)* 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*.



Encoding and decoding: OSPF routing in traffic engineering

(Ericsson et al., 2002; Buriol et al., 2005)

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

Encoding and decoding: OSPF routing in traffic engineering

(Ericsson et al., 2002; Buriol et al., 2005)

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

Encoding and decoding: OSPF routing in traffic engineering

(Ericsson et al., 2002; Buriol et al., 2005)

Encoding

- Solutions are encoded with an *n*-vector *X* of N random keys, where |E| is the number of links. The i-th random key corresponds to the i-th link weight.

Decoder

- INPUT: *n*-vector of random keys
  - For **i = 1, …, |E|**: set **w(i) = ceil (X(i) × wmax)**
  - 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.
- OUTPUT: weights of each link and network congestion cost

Encoding and decoding: Redundant content distribution
(Johnson et al., 2020)

- Given:
  - A directed network G = (V, E);
  - A set of nodes B ∈ V where content-demanding users are located;
  - A set of nodes M ∈ V where content warehouses can be located;
  - The set of all OSPF paths from *m* to *b*, for *m* ∈ M and *b* ∈ B.

- **Compute**:
  - The set of triples $\{ m_1, m_2, b\}^i$, i = 1, 2, ..., T, such that all paths from m_1 to *b* and from m_2 to *b* are disjoint, where $m_1, m_2 \in$ M and *b* ∈ B.

- **Solve the covering by pairs problem**:
  - Find a smallest-cardinality set M* ⊆ M such that for all *b* ∈ B, there exists a triple $\{ m_1, m_2, b\}$ in the set of triples such that $m_1, m_2 \in$ M*.

Encoding and decoding: Redundant content distribution
(Johnson et al., 2020)

Greedy algorithm for covering by pairs

- initialize partial cover M* = { }
- while M* is not a cover do:
  - find $m \in M \setminus M^*$ such that $M^* \cup \{m\}$ covers a maximum number of additional user nodes (break ties by vertex index) and set $M^* = M^* \cup \{m\}$
  - if no $m \in M \setminus M^*$ yields an increase in coverage, then choose a pair $\{m_1, m_2\} \in M \setminus M^*$ that yields a maximum increase in coverage and set $M^* = M^* \cup \{m_1\} \cup \{m_2\}$
  - if no pair exists, then the problem is infeasible

Encoding and decoding: Redundant content distribution (Johnson et al., 2020)

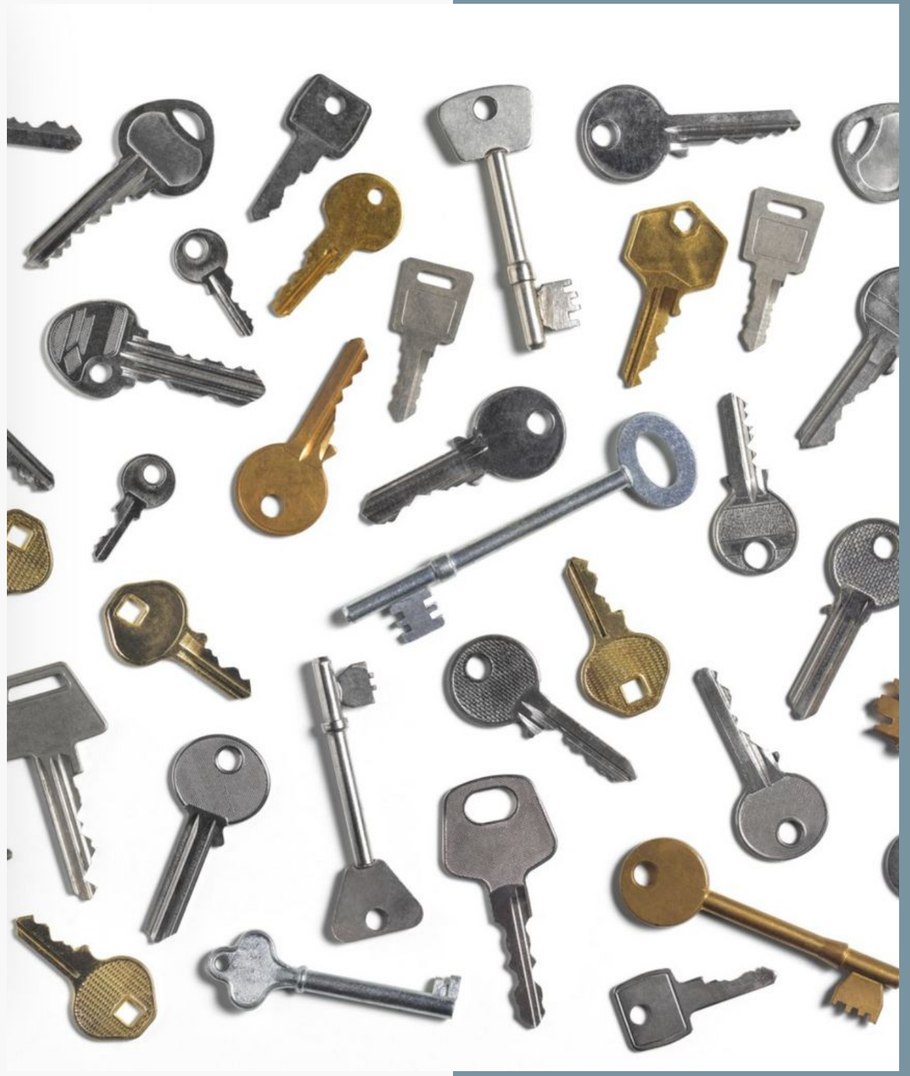
Encoding

- Solutions are encoded with an *n*-vector *X* of |M| random keys, where |M| is the number of potential data warehouse nodes. The i-th random key corresponds to the i-th potential data warehouse node.

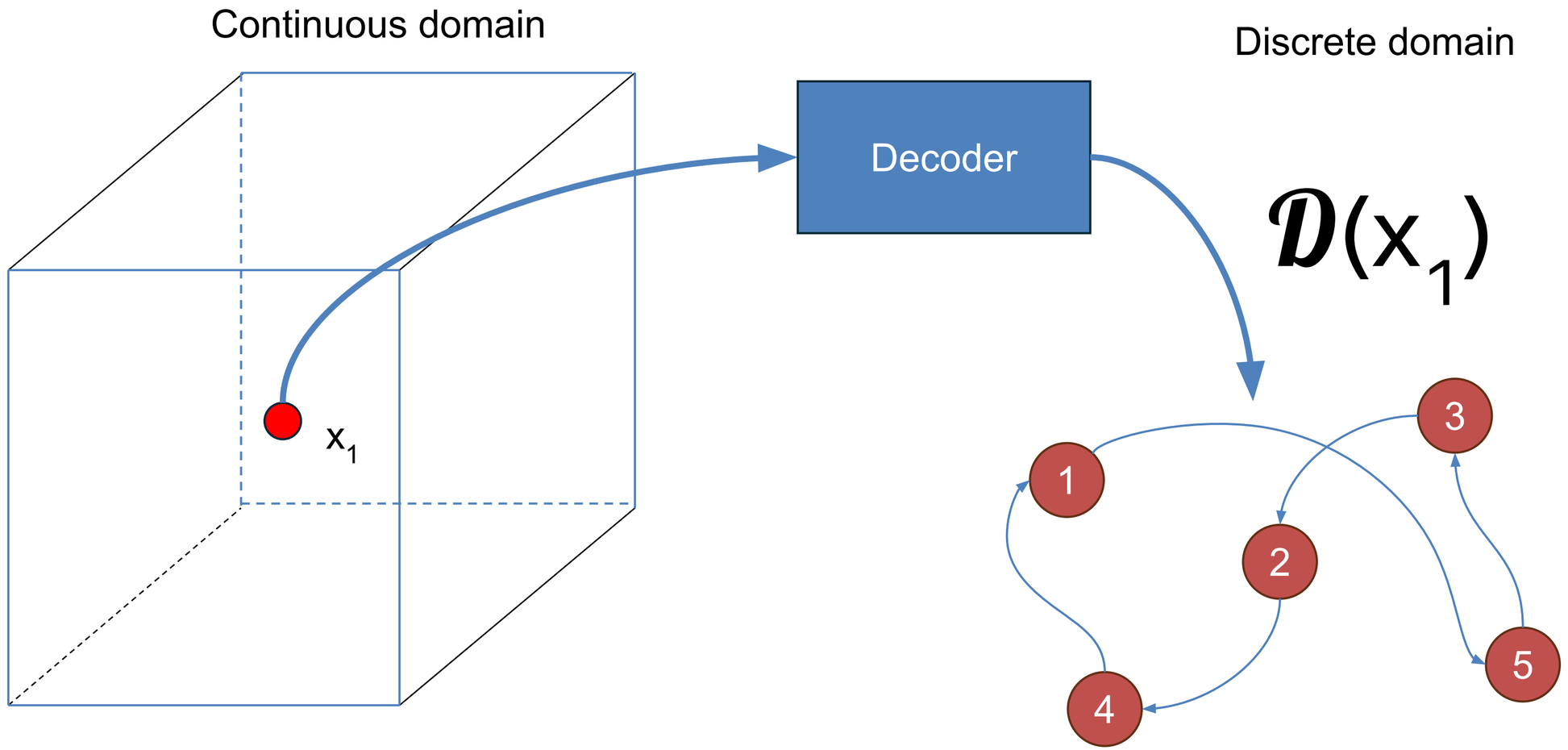
Decoder

- INPUT: *n*-vector of random keys
  - For i = 1, …, n: if X(i) > 0.5, add i-th data warehouse node to solution
  - If solution is feasible, i.e. all users are covered: STOP
  - Else, apply greedy algorithm to cover uncovered user nodes.
  - Remove superfluous elements from cover
- OUTPUT: a cover by pairs $M^* \subseteq M$

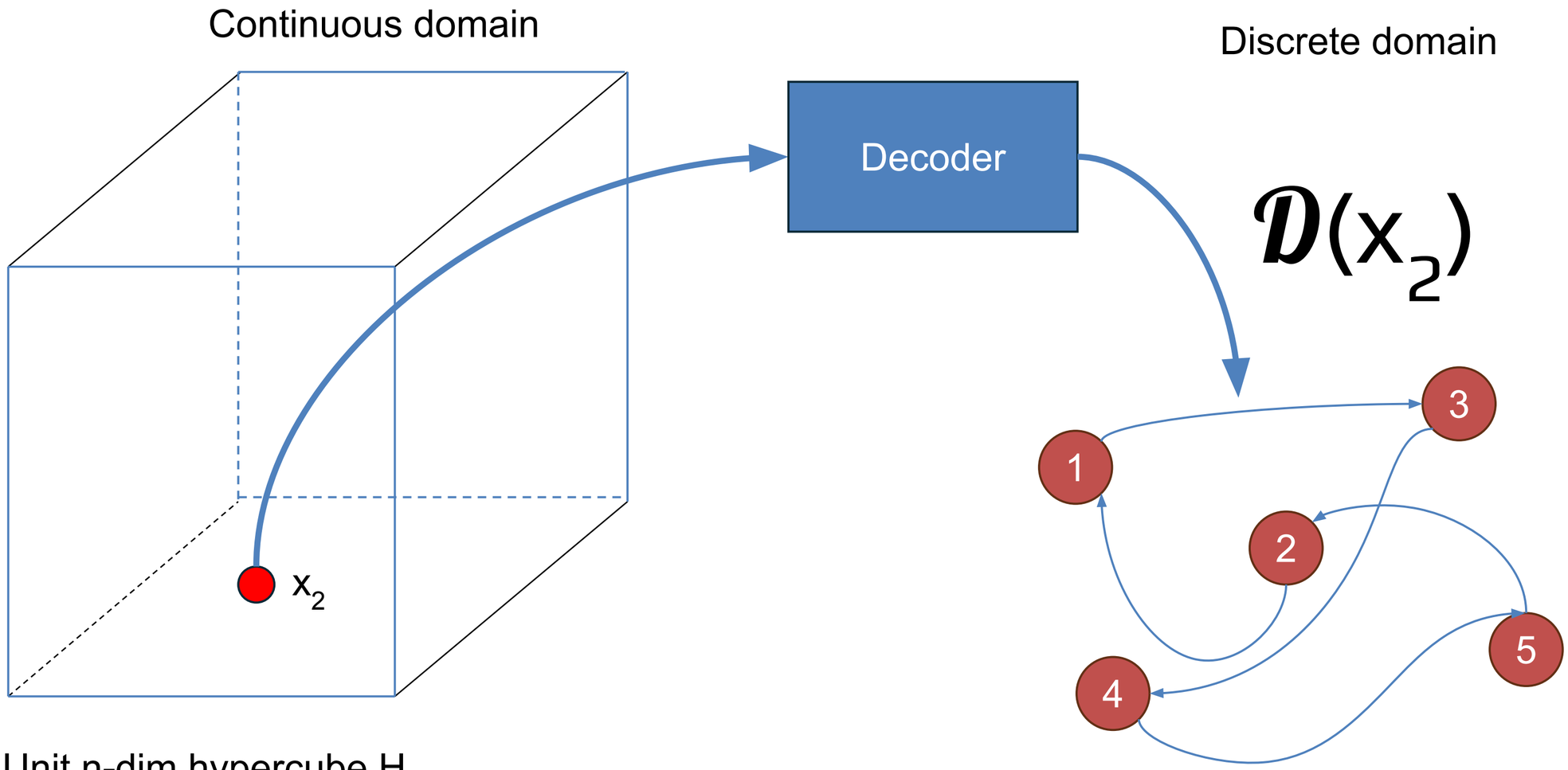
Random-Key Optimizers (RKO)



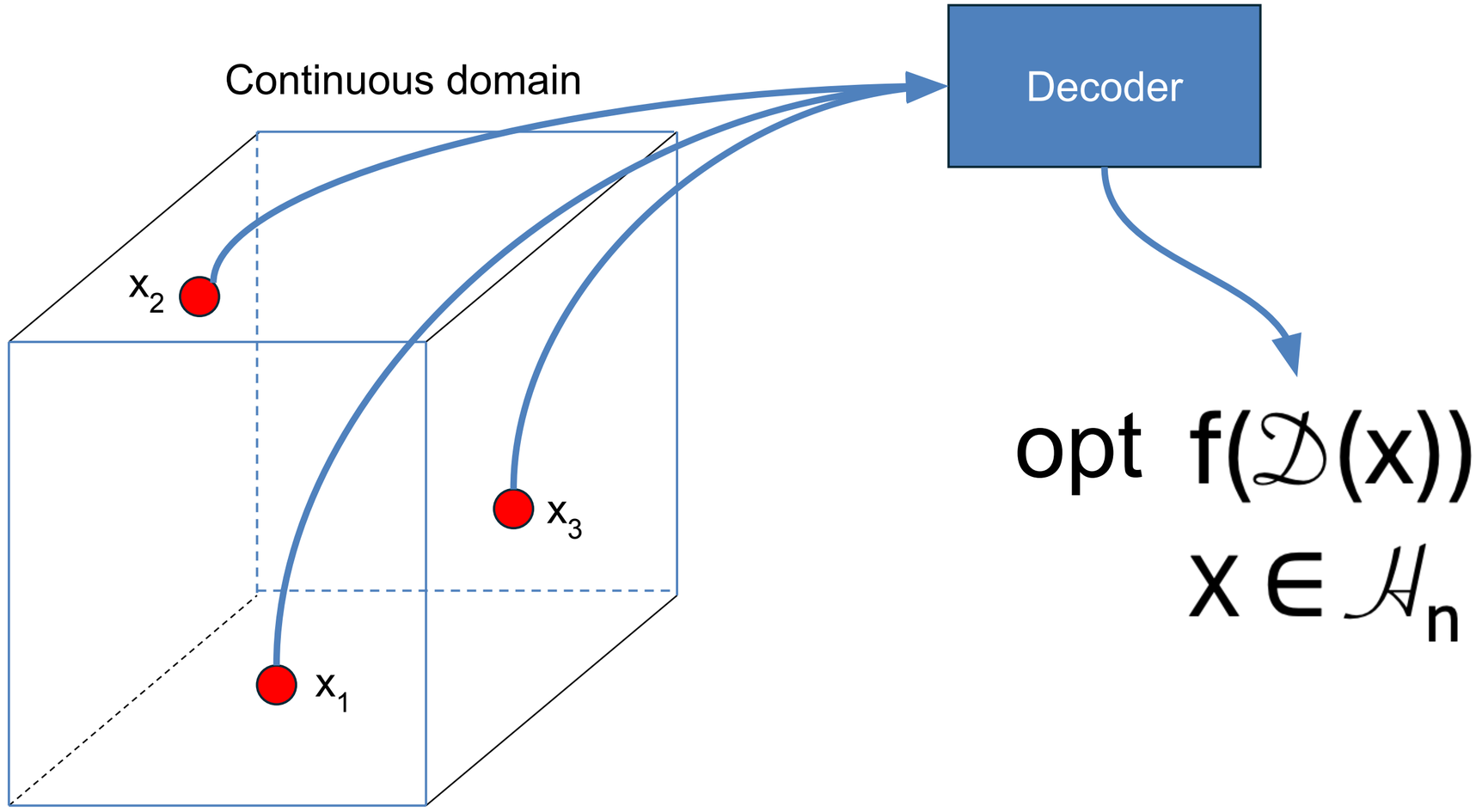
Random-key optimizers



Random-key optimizers

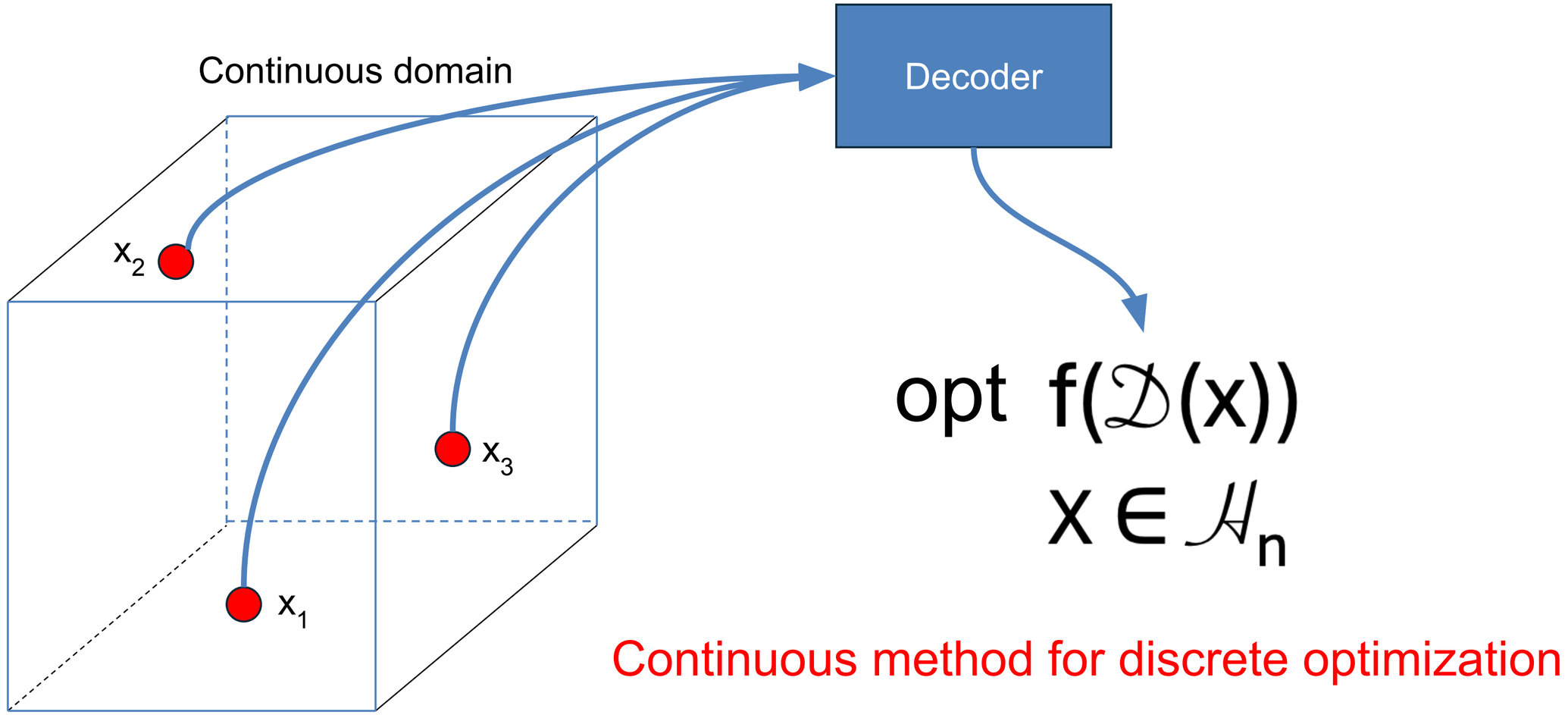


Random-key optimizers



Unit n-dim hypercube H_n

Random-key optimizers

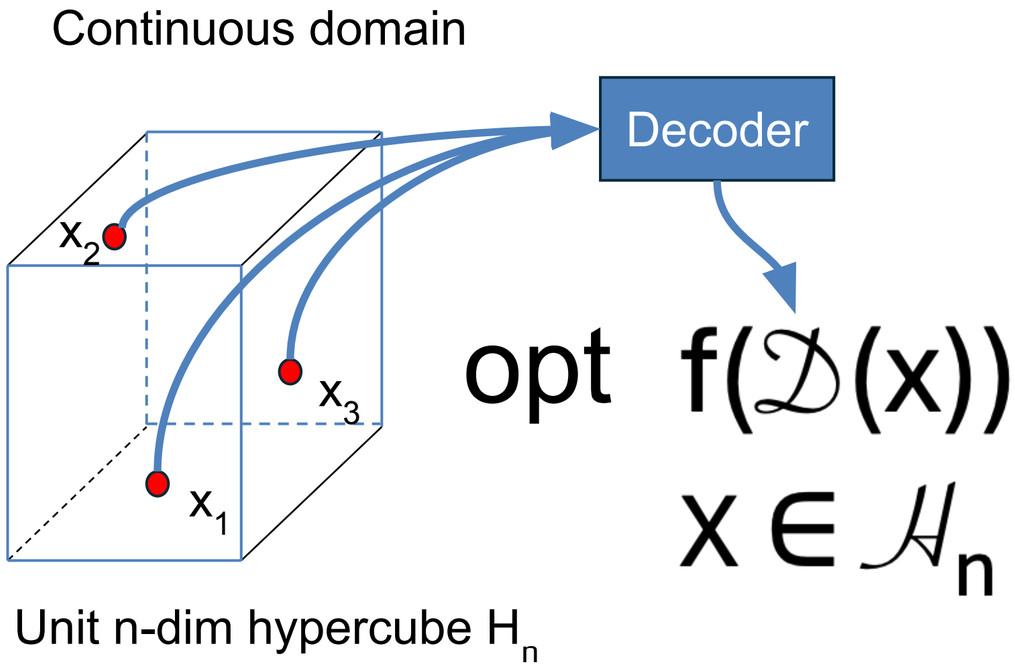


Unit n-dim hypercube H_n

Continuous method for discrete optimization

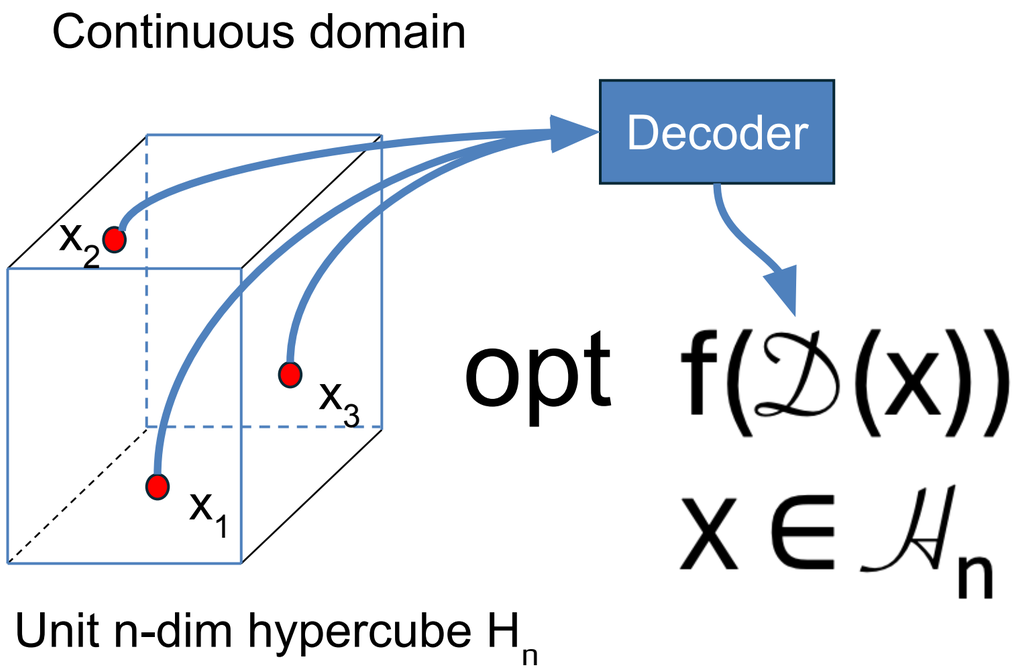
Random-key optimizers

- Random-Key Genetic Algorithms (Bean, 1994)



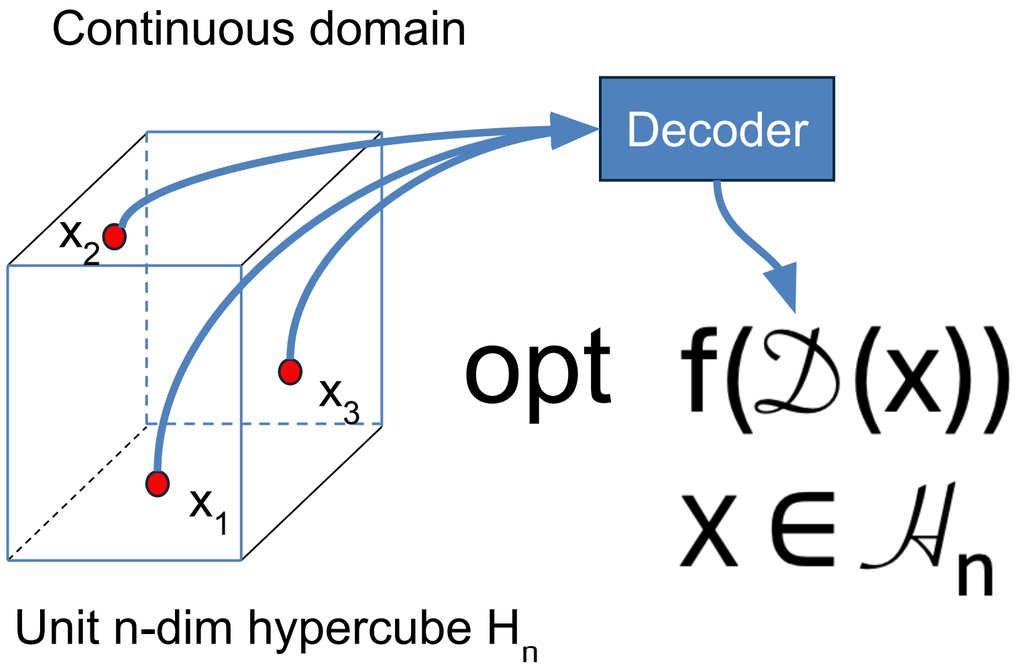
Random-key optimizers

- Random-Key Genetic Algorithms (Bean, 1994)
- Biased Random-Key Genetic Algorithms (Gonçalves & Resende, 2011)



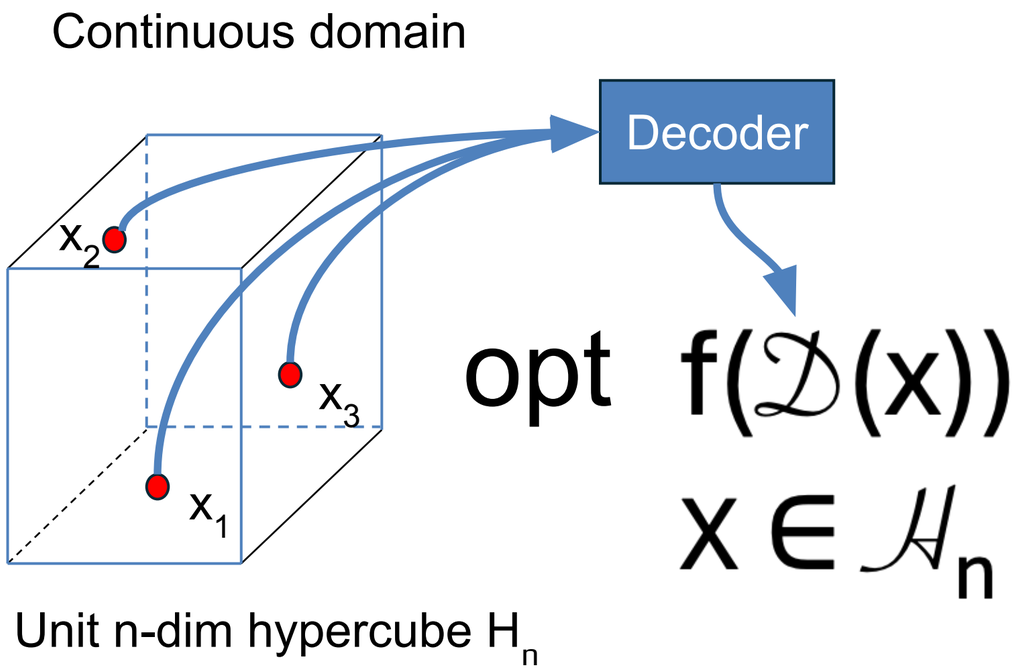
Random-key optimizers

- Random-Key Genetic Algorithms (Bean, 1994)
- Biased Random-Key Genetic Algorithms (Gonçalves & Resende, 2011)
- RKO Dual Annealing for robot motion planning at BMW (Schuetz et al., 2022)



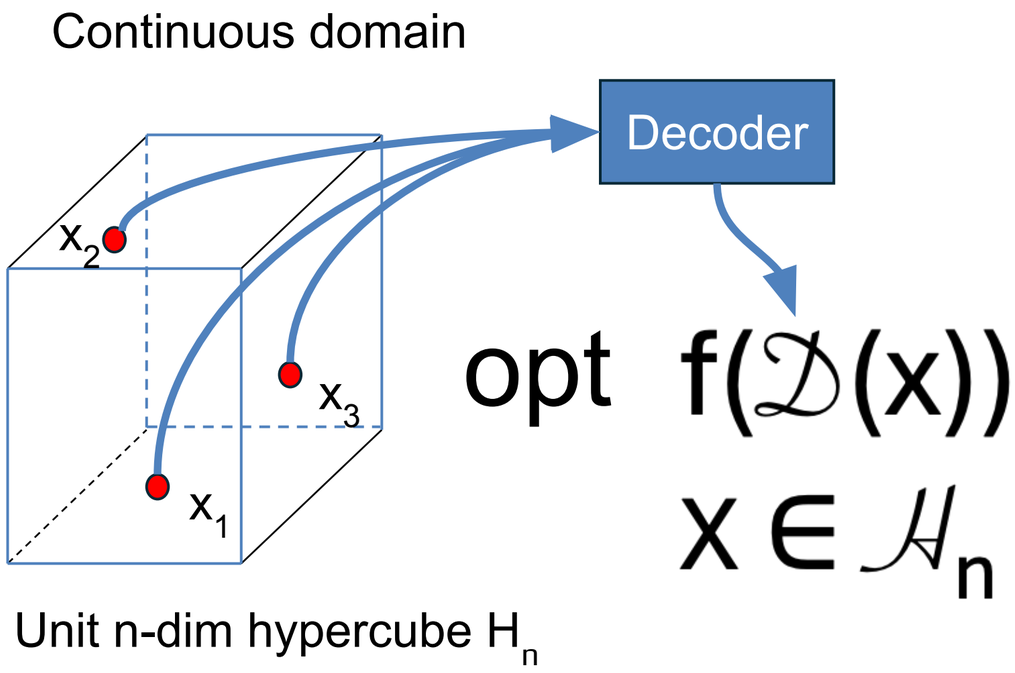
Random-key optimizers

- Random-Key Genetic Algorithms (Bean, 1994)
- Biased Random-Key Genetic Algorithms (Gonçalves & Resende, 2011)
- RKO Dual Annealing for robot motion planning at BMW (Schuetz et al., 2022)
- RKOs based on SA, ILS, and VNS for the tree hub location problem (Mangussi et al., 2023)

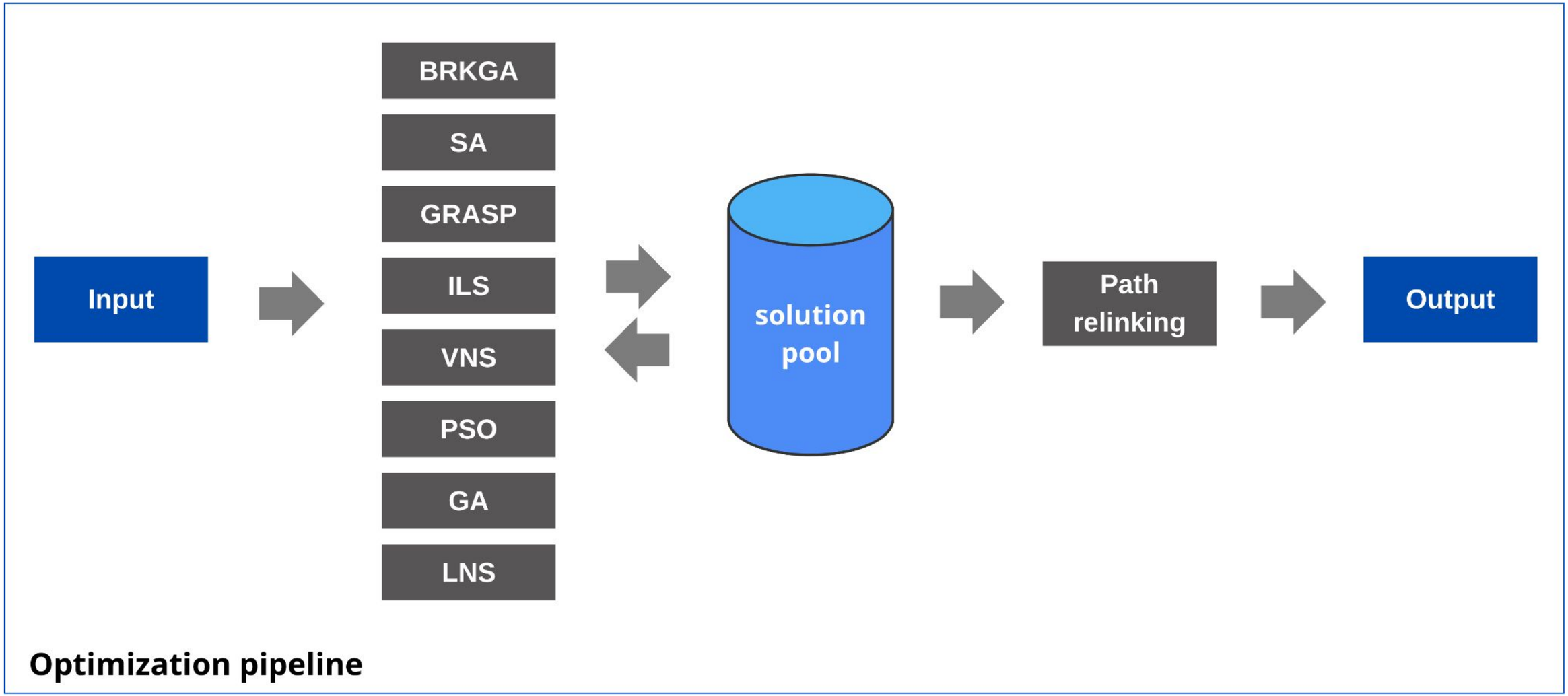


Random-key optimizers

- Random-Key Genetic Algorithms (Bean, 1994)
- Biased Random-Key Genetic Algorithms (Gonçalves & Resende, 2011)
- RKO Dual Annealing for robot motion planning at BMW (Schuetz et al., 2022)
- RKOs based on SA, ILS, and VNS for the tree hub location problem (Mangussi et al., 2023)
- RKO-GRASP (Chaves et al. 2024)



Random-key optimizers



Optimization pipeline

Random-key optimizers

- **RKO Framework**
  - Initial and pool of elite solutions
  - Shaking
  - Blending
  - Local Search
  - Metaheuristics

Initial solutions

- Solutions are represented by n-dimensional vectors $\chi \in [0, 1)^n$, where each random key x_i is randomly generated within the half-open interval [0, 1).

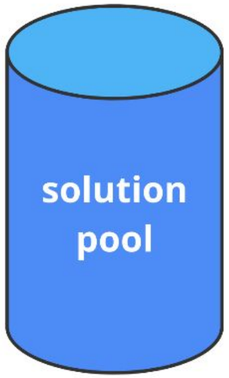
- The quality of each solution is computed by an objective function value, obtained through the application of a problem-specific **decoder** function $\mathcal{D}$ to χ, denoted as $f(\mathcal{D}(\chi))$.

1	2	3	4	5	6
0.8	0.1	0.5	0.2	0.4	0.6

Sort the random keys

2	4	5	3	6	1
0.1	0.2	0.4	0.5	0.6	0.8

Pool of elite solutions



- We maintains a **shared** pool of elite solutions (pool)
- It is initialized with λ randomly generated random-key vectors
- Initial pool is improved by the Farey Local Search heuristic
- Clone solutions (i.e., solutions with identical objective function values) are perturbed by the shaking method
- Solutions generated by a metaheuristic that improves its current best solution is considered for inclusion in the elite pool.
  - absence of a clone within the pool
  - maintain the pool size by removing the solution with the worst objective function value

Shaking

- To modify a random-key vector χ, a perturbation rate β is employed.
- This intensity is randomly generated within an interval [β_{min}, β_{max}]

Algorithm 1: Shaking method

Input: Random-key vector χ, β_{min}, β_{max}
Output: Changed random-key vector χ
Generate shaking rate β randomly within the interval $[\beta_{min}, \beta_{max}]$;
for $k \leftarrow 1$ ***to*** $\beta \times n$ **do**
    Randomly select one shaking move m from $\{1, 2, 3, 4\}$;
    switch m **do**
        case 1 **do**
            Apply Random move in χ;
        end
        case 2 **do**
            Apply Mirror move in χ;
        end
        case 3 **do**
            Apply Swap move in χ;
        end
        case 4 **do**
            Apply Swap Neighbor move in χ;
        end
    end
end
return χ;

Blending

- The blending method creates a new random-key vector by combining two solutions (χ^a and χ^b)
- This process extends the uniform crossover (UX) concept (Davis, 1991), incorporating additional stochastic elements

Algorithm 2: Blending method

Input: Random-key vector χ^a, Random-key vector χ^b, *factor*, ρ, μ
Output: New random-key vector χ^c

for $i \leftarrow 1$ ***to*** n **do**
  if $\mathit{UnifRand}(0,1) < \mu$ **then**
    $\chi_i^c \leftarrow \mathit{UnifRand}(0,1)$;
  end
  else
    if $\mathit{UnifRand}(0,1) < \rho$ **then**
      $\chi_i^c \leftarrow \chi_i^a$
    end
    else
      if $factor = 1$ **then**
        $\chi_i^c \leftarrow \chi_i^b$
      end
      if $factor = -1$ **then**
        $\chi_i^c \leftarrow 1.0 - \chi_i^b$
      end
    end
  end
end
return χ^c;

Local Search

- Random Variable Neighborhood Descent (RVND) algorithm (Penna et al., 2013)
- RVND randomly determines the order of neighborhood heuristics applied in each iteration, thereby efficiently navigating diverse solution spaces.

Algorithm 3: RandomVND

Input: χ
Output: The best solution in the neighborhoods.

```
Initialize the Neighborhood List (NL);
while NL ≠ 0 do
    Choose a neighborhood N^i ∈ NL at random;
    Find the best neighbor χ' of χ ∈ N^i;
    if f(D(χ')) < f(D(χ)) then
        χ ← χ';
        Restart NL;
    else
        Remove N^i from the NL ;
return χ
```

Local Search

- We developed four problem-independent local search (LS) heuristics designed to operate within the random-key solution space.

- These heuristics constitute distinct neighborhood structures integrated into the RVND:
  - Swap LS
  - Mirror LS
  - Farey LS
  - Nelder-Mead LS

Swap Local Search

Algorithm 4: Swap Local Search

Input: Random-key vector χ
Output: Best random-key vector χ^{best} found in the neighborhood

```
1  Define a vector RK with random order for the random-key indices;
2  Update the best solution found χ^best ← χ;
3  for i ← 1 to n − 1 do
4      for j ← i + 1 to n do
5          Swap random keys RK_i and RK_j of χ;
6          if f(D(χ)) < f(D(χ^best)) then
7              χ^best ← χ;
8          else
9              χ ← χ^best;
10 return χ^best;
```

Mirror Local Search

Algorithm 5: Mirror Local Search

Input: Random-key vector χ
Output: Best random-key vector χ^{best} found in the neighborhood

Define a vector RK with random order for the random-key indices;
Update the best solution found $\chi^{best} \leftarrow \chi$;
for $i \leftarrow 1$ ***to*** n **do**
    Set the value of the random key RK_i of χ with its complement;
    if $f(\mathcal{D}(\chi)) < f(\mathcal{D}(\chi^{best}))$ **then**
        $\chi^{best} \leftarrow \chi$;
    else
        $\chi \leftarrow \chi^{best}$;
return χ^{best};

Farey Local Search

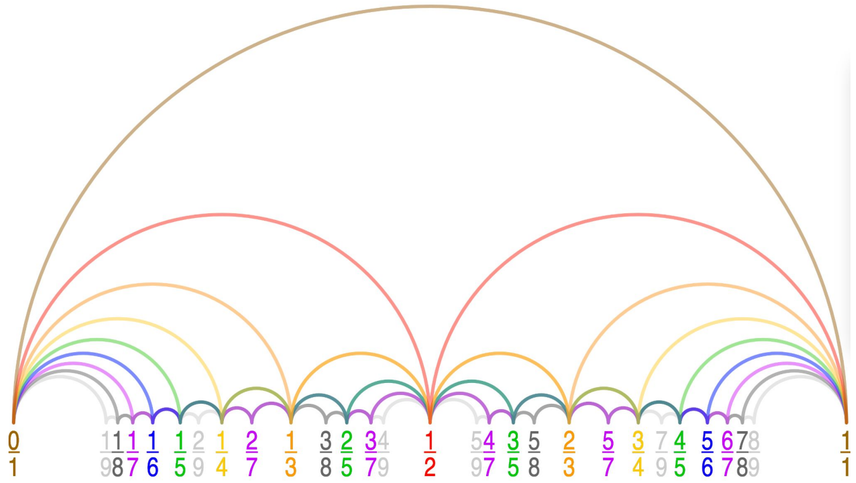
- Adjusts the value of each random key by randomly selecting values between consecutive terms of the Farey sequence (Niven et al., 1991).
- The Farey sequence of order η includes all completely reduced fractions between 0 and 1 with denominators less than or equal to η, arranged in ascending order.

Algorithm 6: Farey Local Search

Input: Random-key vector χ
Output: Best random-key vector χ^{best} found in the neighborhood

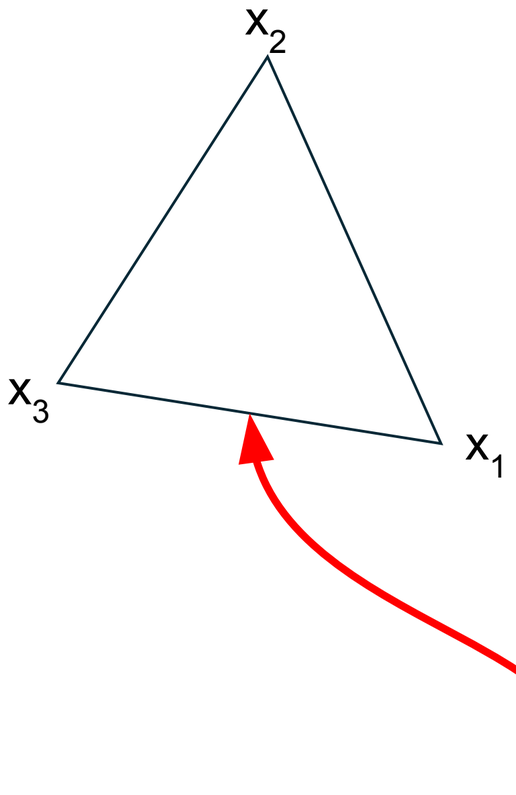
Define a vector RK with random order for the random-key indices;
Update the best solution found $\chi^{best} \leftarrow \chi$;
for $i \leftarrow 1$ ***to*** n **do**
  for $j \leftarrow 1$ ***to*** $|F| - 1$ **do**
    Set the value of the random key RK_i of χ with $\texttt{UnifRand}(F_j, F_{j+1})$;
    if $f(\mathcal{D}(\chi)) < f(\mathcal{D}(\chi^{best}))$ **then**
      $\chi^{best} \leftarrow \chi$;
    else
      $\chi \leftarrow \chi^{best}$;
return χ^{best};

$$F = \left\{ \frac{0}{1}, \frac{1}{7}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{2}{7}, \frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{1}{2}, \frac{4}{7}, \frac{3}{5}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{6}{7}, \frac{1}{1} \right\}$$



Nelder-Mead Local Search

Nelder and Mead (1965)

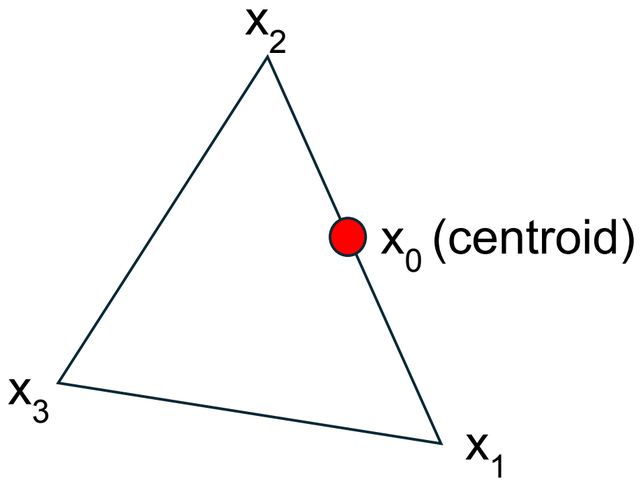


$$f(\mathcal{D}(x_1)) \leq f(\mathcal{D}(x_2)) \leq f(\mathcal{D}(x_3))$$

where one of the solutions is the current solution of the RVND and the other two are elite solutions found during the search.

The three solutions form the N-M simplex

Nelder-Mead Local Search



Five moves:

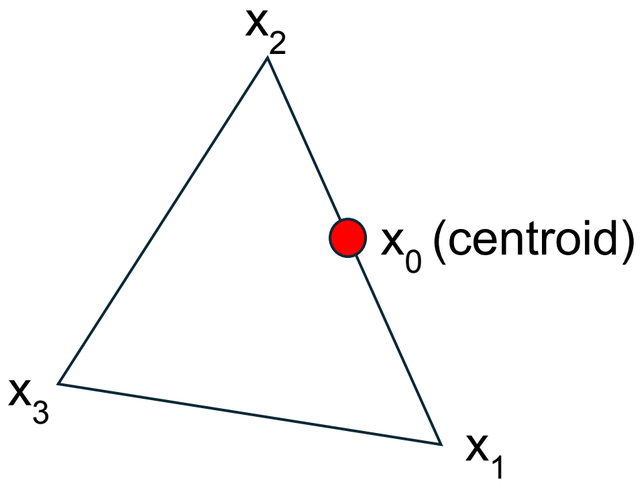
1. Reflection
2. Expansion
3. Inside contraction
4. Outside contraction
5. Shrinking

$x_0 = \text{Blending}(x_1, x_2, 1)$

Instead of using standard Nelder-Mead moves, we use the Blending method between solutions A and B to generate moves (C). There are two types of blendings, Blending(A,B,1) and Blending (A,B,-1) defined by parameter factor:

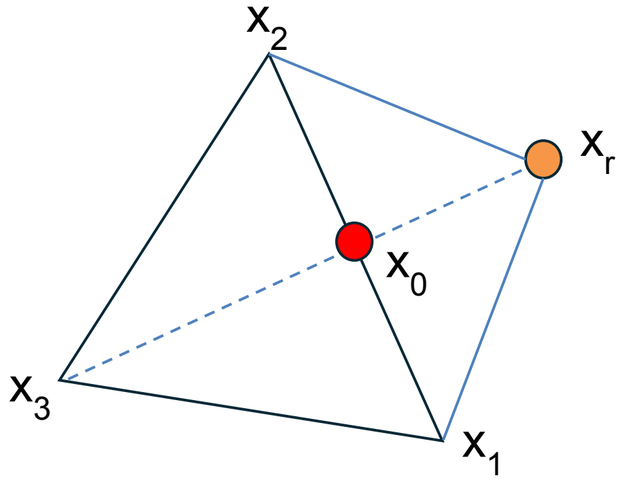
factor = 1	*factor* = −1
A: $a_1\ a_2\ a_3\ a_4$	A: $a_1\ a_2\ a_3\ a_4$
B: $b_1\ b_2\ b_3\ b_4$	B: $b_1\ b_2\ b_3\ b_4$
C: $a_1\ a_2\ b_3\ a_4$	C: $a_1\ a_2\ 1-b_3\ a_4$

Nelder-Mead Local Search



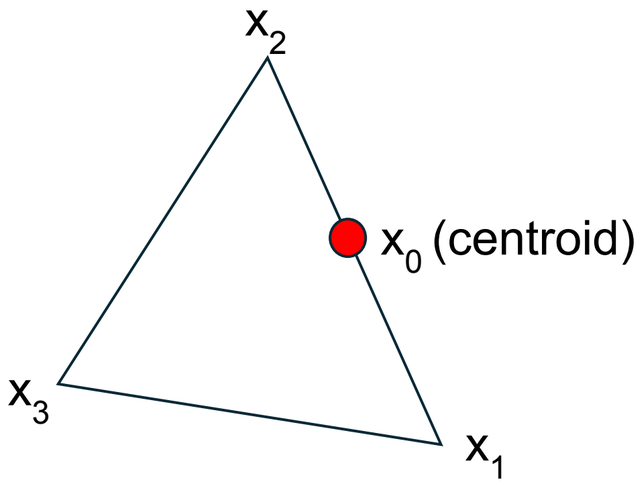
Five moves:

1. Reflection
2. Expansion
3. Inside contraction
4. Outside contraction
5. Shrinking



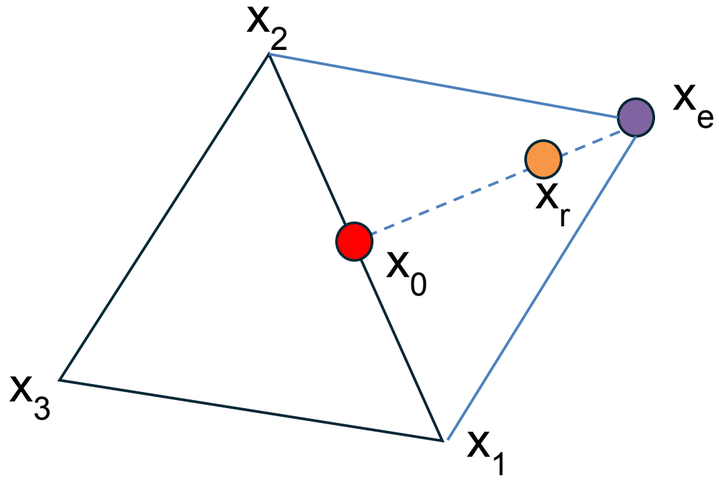
x_r = Blending (x_0, x_3,-1)

Nelder-Mead Local Search



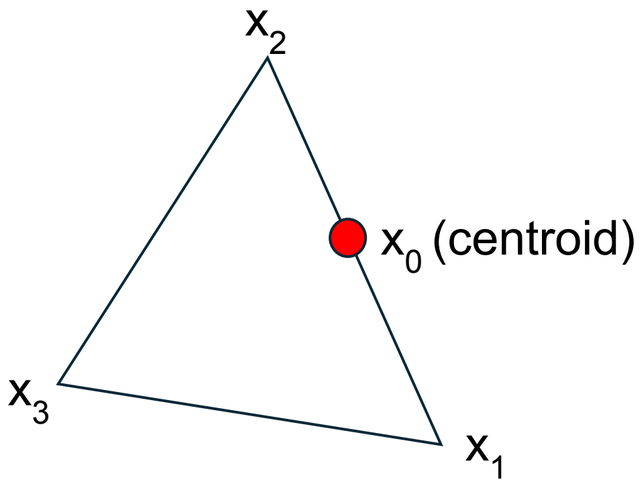
Five moves:

1. Reflection
2. Expansion
3. Inside contraction
4. Outside contraction
5. Shrinking



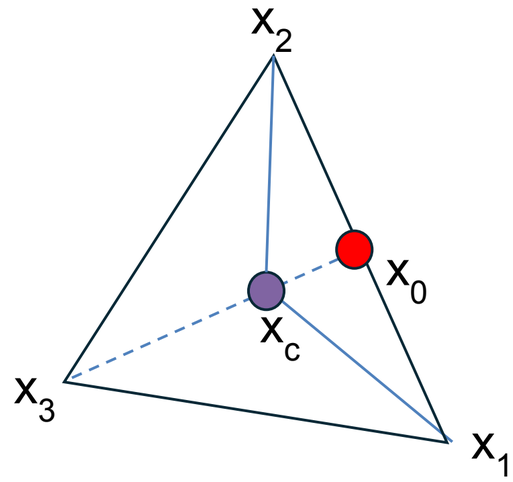
x_e = Blending (x_r, x_0,-1)

Nelder-Mead Local Search



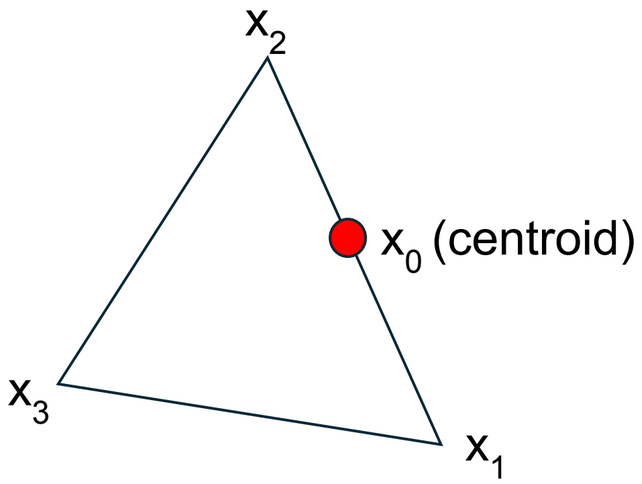
Five moves:

1. Reflection
2. Expansion
3. Inside contraction
4. Outside contraction
5. Shrinking



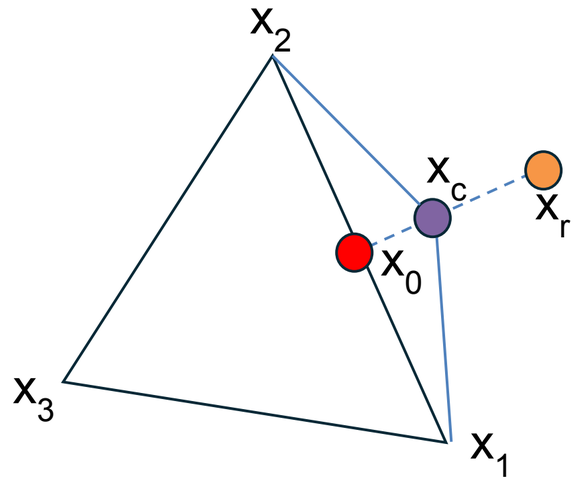
x_c = Blending (x_0, x_3,1)

Nelder-Mead Local Search



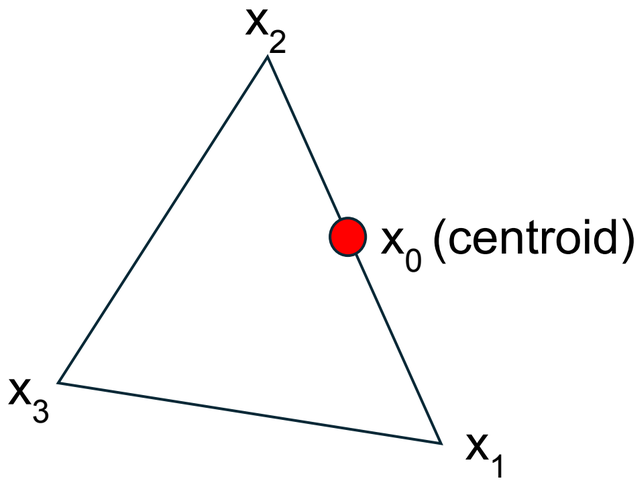
Five moves:

1. Reflection
2. Expansion
3. Inside contraction
4. Outside contraction
5. Shrinking



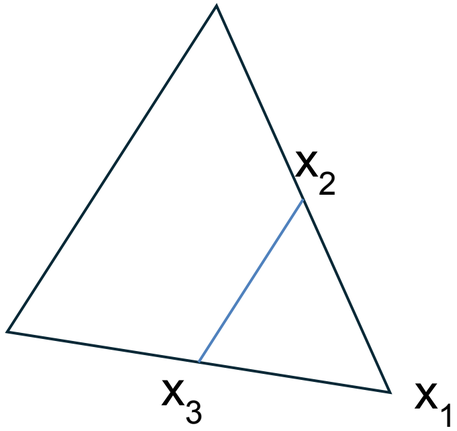
x_c = Blending (x_r,x_0,1)

Nelder-Mead Local Search



Five moves:

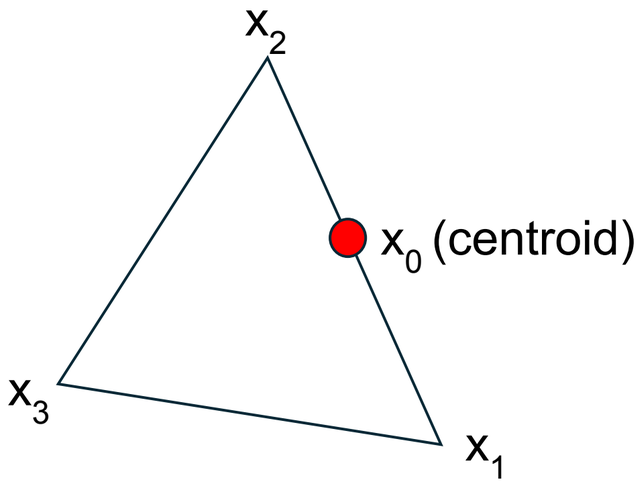
1. Reflection
2. Expansion
3. Inside contraction
4. Outside contraction
5. Shrinking



$x_2 = \text{Blending}(x_1, x_2, 1)$

$x_3 = \text{Blending}(x_1, x_3, 1)$

Nelder-Mead Local Search



Five moves:

1. Reflection
2. Expansion
3. Inside contraction
4. Outside contraction
5. Shrinking

Use Blending to guarantee that generated solution is always in H_n

Nelder-Mead Local Search

Algorithm 7: Nelder-Mead Local Search

Data: χ^1, χ^2, χ^3, n, h
Result: The best solution found in *simplex* X.

Initialize simplex: $X \leftarrow \{\chi^1, \chi^2, \chi^3\}$;
Sort simplex X by objective function value;
Compute the simplex centroid $\chi^0 \leftarrow \texttt{Blending}(\chi^1, \chi^2, 1)$;
$iter \leftarrow 0$;
$numIter \leftarrow n \cdot e^{-2}$;
while $iter < numIter$ **do**
    $shrink \leftarrow 0$;
    $iter \leftarrow iter + 1$;
    Compute reflection solution $\chi^r \leftarrow \texttt{Blending}(\chi^0, \chi^3, -1)$;
    if $f(\mathcal{D}(\chi^r)) < f(\mathcal{D}(\chi^1))$ **then**
        Compute expansion solution $\chi^e \leftarrow \texttt{Blending}(\chi^r, \chi^0, -1)$;
        if $f(\mathcal{D}(\chi^e)) < f(\mathcal{D}(\chi^r))$ **then**
            $\chi^3 \leftarrow \chi^e$;
        else
            $\chi^3 \leftarrow \chi^r$;
    else
        if $f(\mathcal{D}(\chi^r)) < f(\mathcal{D}(\chi^2))$ **then**
            $\chi^3 \leftarrow \chi^r$;
        else
            if $f(\mathcal{D}(\chi^r)) < f(\mathcal{D}(\chi^3))$ **then**
                Compute contraction solution $\chi^c \leftarrow \texttt{Blending}(\chi^r, \chi^0, 1)$;
                if $f(\mathcal{D}(\chi^c)) < f(\mathcal{D}(\chi^r))$ **then**
                    $\chi^3 \leftarrow \chi^c$;
                else
                    $shrink \leftarrow 1$;
            else
                Compute contraction solution $\chi^c \leftarrow \texttt{Blending}(\chi^0, \chi^3, 1)$;
                if $f(\mathcal{D}(\chi^c)) < f(\mathcal{D}(\chi^3))$ **then**
                    $\chi^3 \leftarrow \chi^c$;
                else
                    $shrink \leftarrow 1$;
    if $shrink = 1$ **then**
        Replace all solutions except the best χ^1 with $\chi^i \leftarrow \texttt{Blending}(\chi^1, \chi^i, 1), i = 2, 3$;
    Sort simplex X by objective function value;
    Compute the simplex centroid $\chi^0 \leftarrow \texttt{Blending}(\chi^1, \chi^2, 1)$;
return χ^1

Metaheuristics

- We explore the adaptability of RKO by implementing a comprehensive framework that incorporates eight distinct metaheuristics
  - biased random-key genetic algorithm (**BRKGA**)
  - simulated annealing (**SA**)
  - greedy randomized adaptive search procedure (**GRASP**)
  - iterated local search (**ILS**)
  - variable neighborhood search (**VNS**)
  - particle swarm optimization (**PSO**)
  - genetic algorithm (**GA**)
  - large neighborhood search (**LNS**)
  - BRKGA with clustering search (**BRKGA-CS**)

BRKGA (Gonçalves and Resende, 2011)

Algorithm 8: BRKGA

Data: *time_limit*
Output: Best solution found χ^{best}

Randomly generate the population P with p solutions;
Evaluate and sort P by the objective function. Store the best solution in χ^{best};
while *time_limit is not reached* **do**
    Classify P as elite or non-elite solutions;
    Create elite set P_e using p_e as a guide;
    Create the offspring set P_c through the Blending method, using ρ_e and $factor = 1$ as guides;
    Create the mutant set P_m using p_m as guide;
    $P \leftarrow P_e \cup P_c \cup P_m$;
    Evaluate and sort P by the objective function;
    if *the best solution improved* **then**
        $\chi^{best} \leftarrow \text{RVND}(P^1)$;
        Store the best solution in χ^{best};
return χ^{best};

GA (Holland, 1992; Goldberg, 1989)

Algorithm 9: GA

Data: *time_limit*
Output: Best solution found χ^{best}

```
1  Randomly generate the population P with p solutions;
2  Evaluate P by the objective function. Store the best solution in χ^best;
3  while time_limit is not reached do
4      Select parents with the tournament method;
5      Create P' recombining pairs of parents with the Blending method with
         probability p_c, using ρ_e = 0.5, factor = 1, and μ as guides;
6      P ← P';
7      Evaluate P by the objective function;
8      Randomly select a solution χ^i from P;
9      P^i ← RVND(χ^i);
10     if the best solution improved then
11         Store the best solution in χ^best;
12 return χ^best;
```

SA (Kirkpatrick et al., 1983)

Algorithm 10: SA

Data: *time_limit*
Result: Best solution found χ^{best}

Randomly generate an initial solution χ;
$\chi^{best} \leftarrow \chi$, $T \leftarrow T_0$;
while *time_limit is not reached* **do**
    $iter \leftarrow 0$;
    while $iter < SA_{max}$ **do**
        $\chi' \leftarrow \text{Shaking}(\chi, \beta_{min}, \beta_{max})$;
        Calculate the energy difference $\Delta E \leftarrow f(\mathscr{D}(\chi')) - f(\mathscr{D}(\chi))$;
        if $\Delta E \leq 0$ **then**
            $\chi \leftarrow \chi'$;
            if $f(\mathscr{D}(\chi)) < f(\mathscr{D}(\chi^{best}))$ **then**
                $\chi^{best} \leftarrow \chi$;
        else
            Calculate the acceptance probability $\tau \leftarrow \exp\left(-\frac{\Delta E}{T}\right)$;
            if $UnifRand(0, 1) < \tau$ **then**
                $\chi \leftarrow \chi'$;
        $iter \leftarrow iter + 1$;
    $\chi \leftarrow \text{RVND}(\chi)$;
    Update temperature $T \leftarrow \alpha \times T$;
return χ^{best};

C-GRASP (Hirsch et al., 2007)

Algorithm 11: GRASP

Data: *time_limit*
Result: Best solution found (χ^{best})

```
1  Randomly generate an initial solution χ;
2  χ^best ← χ;
3  while time_limit is not reached do
4      h ← hs;
5      while h ≥ he do
6          χ' ← ConstructGreedyRandomized(χ, h);
7          χ'' ← RVND(χ');
8          if f(D(χ'')) < f(D(χ^best)) then
9              χ^best ← χ'';
10         else
11             h ← h/2;
12         if accept(χ'', χ) then
13             χ ← χ'';
14     Randomly generate a new solution χ;
15 return χ^best;
```

https://doi.org/10.23952/jnva.8.2024.6

C-GRASP (Hirsch et al., 2007)

Algorithm 12: ConstructGreedyRandomized

Data: χ, h
Result: A constructive semi-greedy solution.

$UnFixed \leftarrow \{1, 2, ..., n\}$;
$\alpha \leftarrow$ `UnifRand(0,1)`;
$Reuse \leftarrow false$;
while $UnFixed \neq \emptyset$ **do**
  $min \leftarrow +\infty$; $max \leftarrow -\infty$;
  for $i = 1, ..., n$ **do**
    if $i \in UnFixed$ **then**
      if $Reuse = false$ **then**
        $[r_i, g_i] \leftarrow \text{LineSearch}(\chi, h, i)$;
      if $min > g_i$ **then**
        $min \leftarrow g_i$;
      if $max < g_i$ **then**
        $max \leftarrow g_i$;
  $RCL \leftarrow \emptyset$;
  for $i = 1, ..., n$ **do**
    if $i \in UnFixed$ *and* $g_i \leq min + \alpha \cdot (max - min)$ **then**
      $RCL \leftarrow RCL \cup \{i\}$;
  $j \leftarrow \text{RandomlySelectElement}(RCL)$;
  if $\chi_j = r_j$ **then**
    $Reuse \leftarrow true$;
  else
    $\chi_j \leftarrow r_j$;
    $f(\mathcal{D}(\chi)) \leftarrow g_j$;
    $Reuse \leftarrow false$;
  $UnFixed \leftarrow UnFixed \backslash \{j\}$;
return χ

ILS (Lourenço et al., 2003)

Algorithm 13: ILS

Data: *time_limit*
Result: Best solution found (χ^{best})

Randomly generate an initial solution χ^0;
$\chi \leftarrow \text{RVND}(\chi^0)$;
$\chi^{best} \leftarrow \chi$;
$iterNoImprov \leftarrow 0$;
while *time_limit is not reached* **do**
  $\chi' \leftarrow \text{Shaking}(\chi, \beta_{min}, \beta_{max})$;
  $\chi'^* \leftarrow \text{RVND}(\chi')$;
  if *acceptance criterion* (χ'^*, χ) **then**
    $\chi \leftarrow \chi'^*$;
    if $(f(\mathcal{D}(\chi'^*)) < f(\mathcal{D}(\chi^{best}))$ **then**
      $\chi^{best} \leftarrow \chi'^*$;
  else
    $iterNoImprov \leftarrow iterNoImprov + 1$;
  if *history(iterNoImprov)* **then**
    Randomly generate an solution χ;
return χ^{best};

VNS (Mladenovic and Hansen, 1997)

Algorithm 14: VNS

Data: *time_limit*
Result: Best solution found (χ^{best})
Randomly generate an initial solution χ;
$\chi^{best} \leftarrow \chi$;
$iterNoImprov \leftarrow 0$;
while *time_limit is not reached* **do**
  $k \leftarrow 1$;
  while $k < k_{max}$ **do**
    $\beta \leftarrow k \times \beta_{min}$;
    $\chi' \leftarrow \text{Shaking}(\chi, \beta, \beta)$;
    $\chi'^{*} \leftarrow \text{RVND}(\chi')$;
    if *acceptance criterion* $(\chi'^{*}, \chi))$ **then**
      $\chi \leftarrow \chi'^{*}$;
      if $(f(\mathcal{D}(\chi'^{*})) < f(\mathcal{D}(\chi^{best}))$ **then**
        $\chi^{best} \leftarrow \chi'^{*}$;
    else
      $k \leftarrow k + 1$;
      $iterNoImprov \leftarrow iterNoImprov + 1$;
    if *history(iterNoImprov)* **then**
      Randomly generate an solution χ;
return χ^{best};

PSO (Kennedy and Eberhart, 1995)

Algorithm 15: PSO

Data: *time_limit*
Output: Best solution found χ^{best}

```
1  Randomly generate the locations χ^i and velocity v^i of p solutions;
2  Evaluate χ^i. Store the best solution in χ^best;
3  while time_limit is not reached do
4      Generate new velocity v^i for all solutions and dimensions using
         w, r_1, r_2, c_1, c_2 as guides;
5      Calculate new locations χ^i ← χ^i + v^i;
6      Evaluate objective functions at new locations χ^i;
7      Find the current best for each solution χ^i*;
8      if the best solution improved then
9          Store the best solution in χ^best;
10     Randomly select a solution χ^j;
11     χ^j ← RVND(χ^j);
12 return χ^best;
```

LNS (Ropke and Pisinger, 2006)

Algorithm 16: LNS

Data: *time_limit*
Result: Best solution found χ^{best}
Randomly generate an initial solution χ;
$\chi^{best} \leftarrow \chi$, $T \leftarrow T_0$;
while *time_limit is not reached* **do**
  while $T > 0.0001$ **do**
    remove $\beta = UnifRand(\beta_{min}, \beta_{max})$ random keys from χ;
    repair β random keys of χ using the Farey LS;
    $\chi' \leftarrow \text{RVND}(\chi)$;
    Calculate the energy difference $\Delta E \leftarrow f(\mathscr{D}(\chi')) - f(\mathscr{D}(\chi))$;
    if $\Delta E \leq 0$ **then**
      $\chi \leftarrow \chi'$;
      if $f(\mathscr{D}(\chi)) < f(\mathscr{D}(\chi^{best}))$ **then**
        $\chi^{best} \leftarrow \chi$;
    else
      Calculate the acceptance probability $\tau \leftarrow \exp\left(-\frac{\Delta E}{T}\right)$;
      if $UnifRand(0, 1) < \tau$ **then**
        $\chi \leftarrow \chi'$;
    Update temperature $T \leftarrow \alpha \times T$;
  Reanelling $T \leftarrow T_0 \times 0.3$;
return χ^{best};

BRKGA-CS (Chaves and Lorena, 2021)

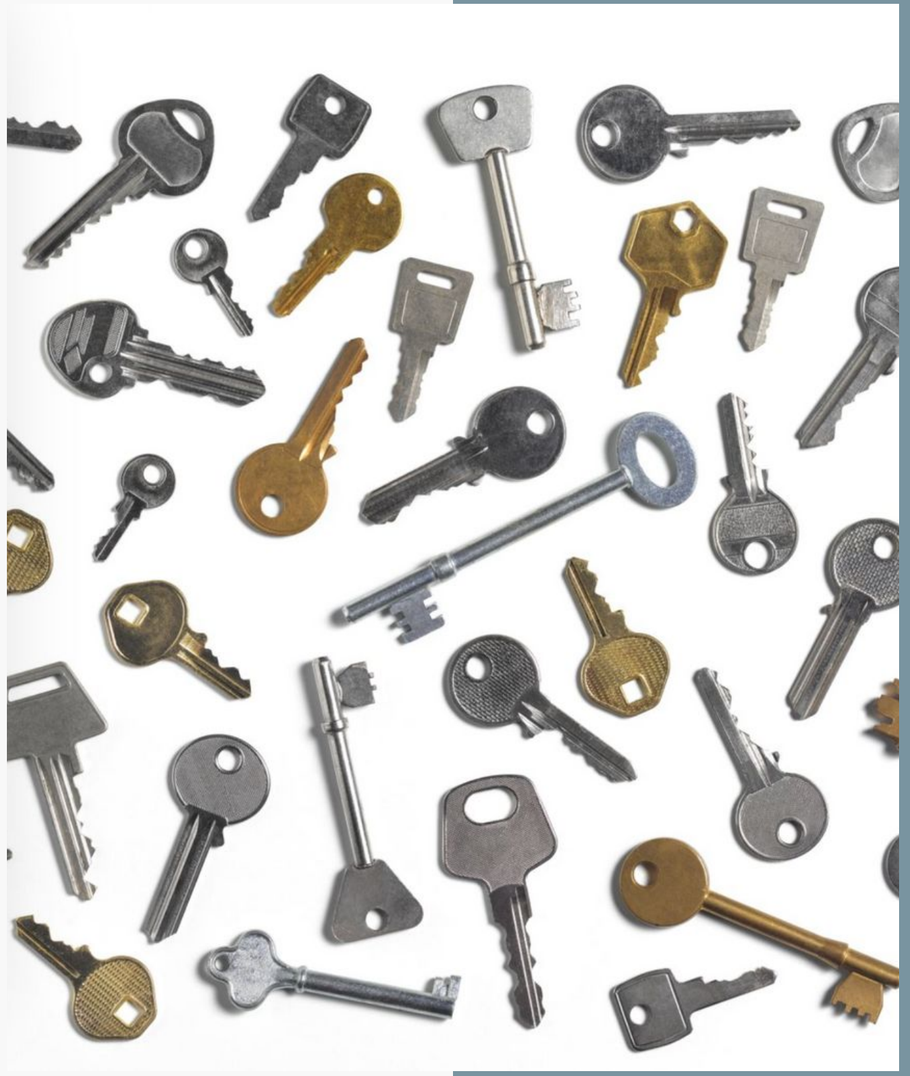
Algorithm 17: BRKGA-CS

Data: *time_limit*
Output: Best solution found χ^{best}

Initialize Q-Table values;
Randomly generate the population P with p solutions;
Evaluate and sort P by the objective function. Store the best solution in χ^{best};
while *time_limit is not reached* **do**
    Set Q-Learning parameters (ϵ, lf, df);
    Choose an action for each parameter (p, p_e, μ, ρ_e) from the Q-Table using the ϵ-greedy policy;
    Classify P as elite or non-elite solutions;
    Create elite set P_e using p_e as a guide;
    Create the offspring set P_c through the blending procedure, using ρ_e, μ, and $factor = 1$ as guides;
    $P \leftarrow P_e \cup P_c$;
    Evaluate and sort P by the objective function;
    if *the best solution improved* **then**
        Store the best solution in χ^{best};
        Set reward (R^i) and update Q-Table;
    if *exploration or stagnation is detected* **then**
        Identify communities in P_e with the clustering method;
        Apply RVND in the best solutions of these communities;
        Apply Shaking in other solutions;
return χ^{best};

Travelling Thief Problem

Challenge



Travelling Thief Problem - Challenge

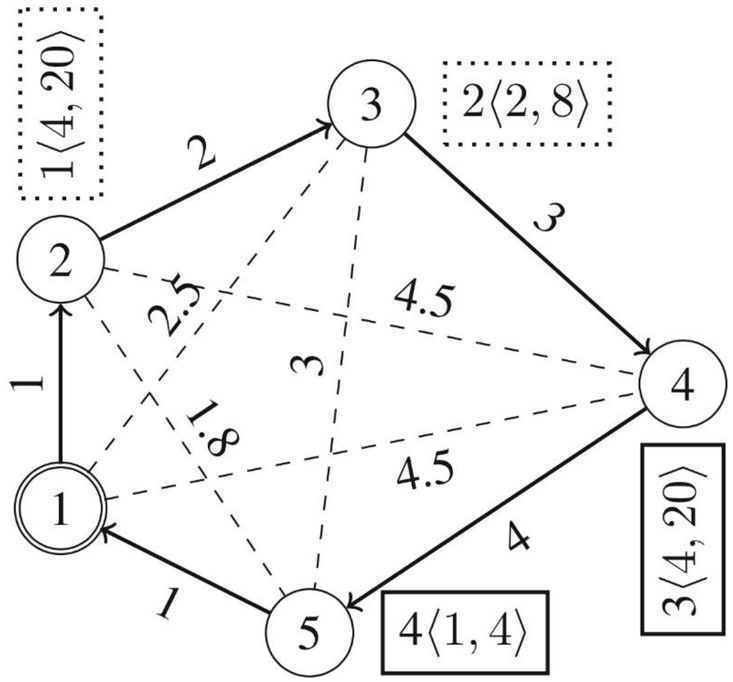
https://mauricio.resende.info/doc/RKO_ver_2024-11-02.pdf

https://github.com/antoniochaves19/RKO

https://sites.google.com/view/ttp-gecco2024/

Travelling Thief Problem - Challenge

Namazi, M., Newton, M. H., Sanderson, C., & Sattar, A. (2023). Solving travelling thief problems using coordination based methods. *Journal of Heuristics*, *29*(4), 487-544.



Knapsack Capacity $W = 6$

Knapsack Renting Rate $R = 1$

Maximum Speed $s_{\max} = 1$

Minimum Speed $s_{\min} = 0.1$

Cyclic Tour $t = \langle 1, 2, 3, 4, 5, 1 \rangle$

Collection plan $p = \langle 0, 0, 1, 1 \rangle \equiv \{3, 4\}$

For tour segment $\langle 1, 2, 3, 4 \rangle$, $s = s_{\max} = 1$

For tour segment $\langle 4, 5 \rangle$, $s = 1 - \frac{4 \times 0.9}{6} = 0.4$

For tour segment $\langle 5, 1 \rangle$, $s = 1 - \frac{5 \times 0.9}{6} = 0.25$

Total Time $T(t, p) = \frac{1+2+3}{1} + \frac{4}{0.4} + \frac{1}{0.25} = 20$

Total Rent $R(t, p) = 20 \times R = 20$

Total Weight $W(p) = 4 + 1 = 5$

Total Profit $P(p) = 20 + 4 = 24$

Net Profit $N(t, p) = 24 - 20 = 4$

Fig. 1 Left: a TTP instance having 5 cities (circles) and 4 items (rectangles) and a TTP solution with the travelled cyclic path (solid lines) and collected items (solid rectangles). Eacy city has a city index. City 1 (double circle) is the designated city to start from and end to. Each item has an item index and a tuple for weight and profit. Lines have distances as labels. Dashed lines are not in the travelled path and dotted rectangles are for items not collected. Right: other required parameters of the TTP instances along with the calculation of the net profit for the TTP solution

Travelling Thief Problem - Challenge

Instances:

https://cs.adelaide.edu.au/~optlog/CEC2014COMP_InstancesNew/?authuser=1

lin105-ttp/lin105_n104_uncorr_05.ttp (2024 best: 9676.764864)
lin105-ttp/lin105_n312_bounded-strongly-corr_02.ttp (2024 best: 24227.32016)
lin105-ttp/lin105_n520_uncorr_01.ttp (2024 best : 26578.57788)

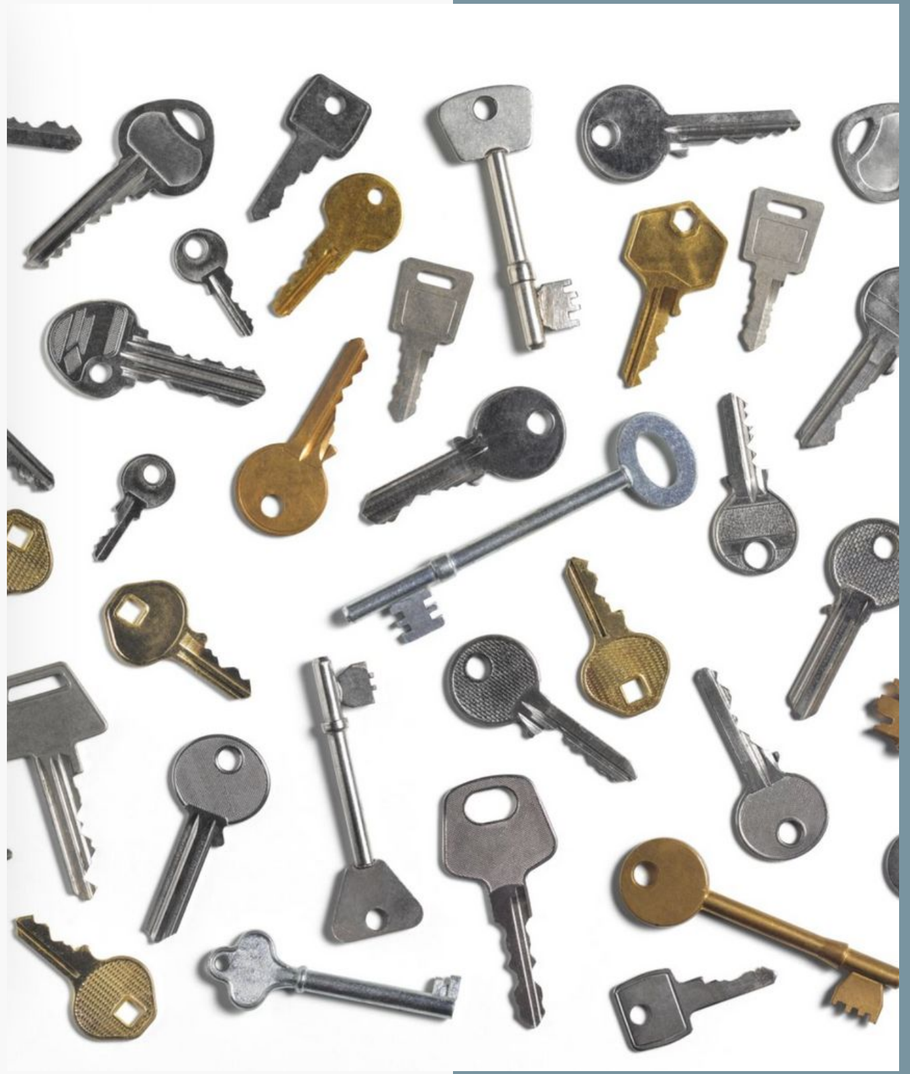
Problem.h:

https://github.com/antoniochaves19/TTP

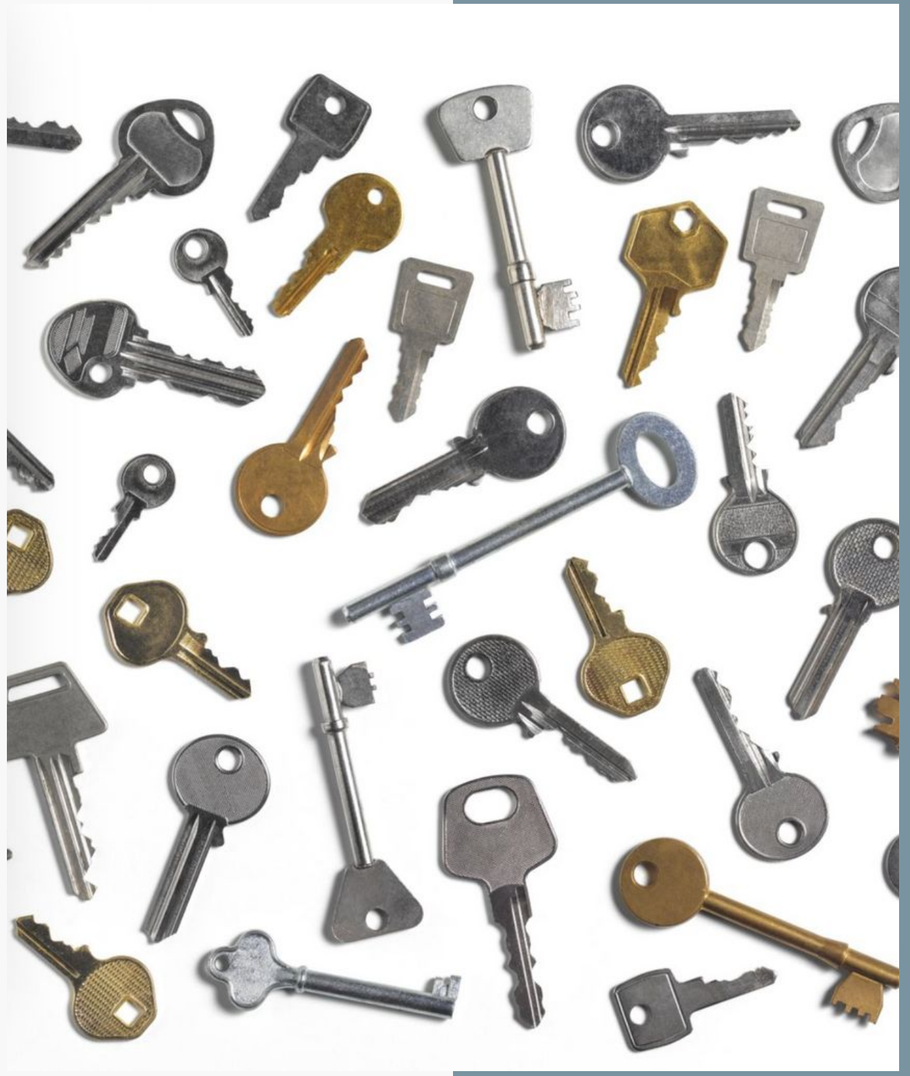
Part II

Contents:

- Examples of encoding and decoding with random keys
- RKO applications in discrete optimization problems
- API

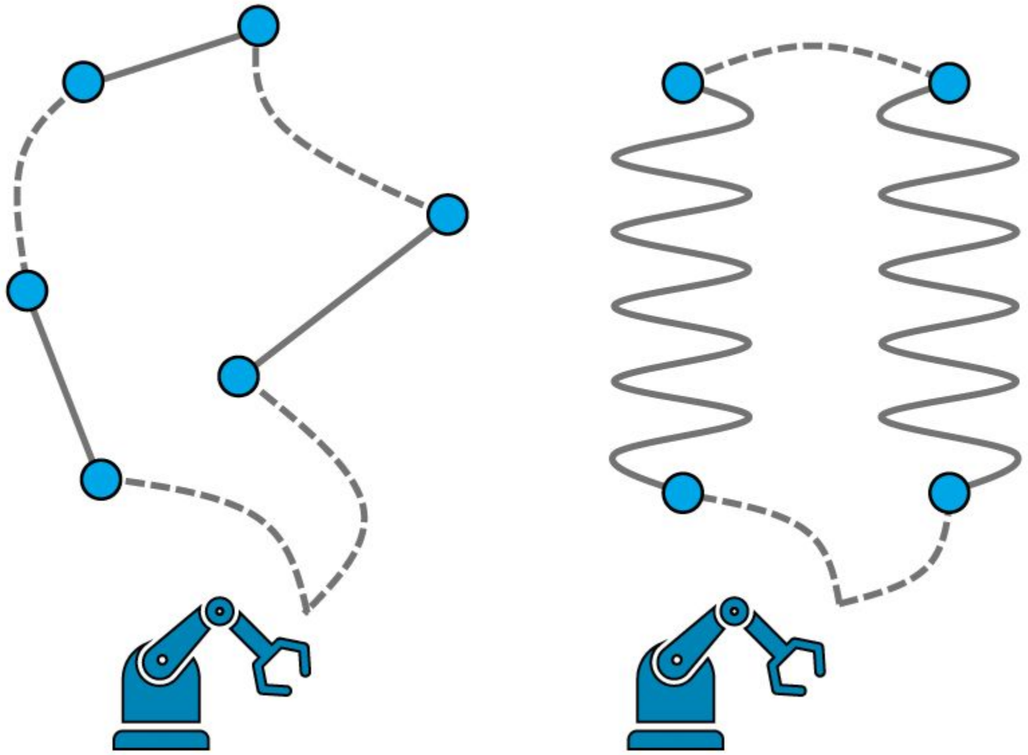


Examples of encoding and decoding with random keys



Robot motion planning

(Schuetz et al., 2022)



In automobile assembly PVC sealant is applied post-welding to every joint to ensure the car body's water-tightness.

Strips of PVC applied are called seams. Modern plants, like BMW, usually deploy a fleet of robots to apply the PVC sealant.

Motion planning involves determination of sequence of joints on which to apply seams and which of several available nozzles to use.

Travel time between $\text{joint}_{(i)}$ to $\text{joint}_{(j)}$ depends not only on the location of each joint but also on which nozzle is used to apply the sealant on each joint and other factors.

The goal is to identify collision-free trajectories such that all seams get processed within the minimum time span.

Robot motion planning

(Schuetz et al., 2022)

TABLE I. Example data set (cost matrix) for illustration. The problem is specified in terms of cost values (in seconds) for pairs of nodes, with every node described by a tuple $[s, d, t, c, p]$ that captures the seam index s, direction d, tool t, tool configuration c, and robot position p. Further details are provided in the main text.

node (from)					node (to)					cost
s	d	t	c	p	s	d	t	c	p	w [s]
0	0	0	0	0	18	1	1	1	1	0.877
11	2	1	0	1	12	1	2	0	1	0.473
11	2	1	0	1	12	0	3	0	0	0.541
32	2	1	2	1	25	2	1	2	1	0.558
⋮					⋮					⋮

Robot home position is [0,0,0,0,0]

In automobile assembly PVC sealant is applied post-welding to every joint to ensure the car body's water-tightness.

Strips of PVC applied are called seams. Modern plants, like BMW, usually deploy a fleet of robots to apply the PVC sealant.

Motion planning involves determination of sequence of joints on which to apply seams and which of several available nozzles to use.

Travel time between joint$_{(i)}$ to joint$_{(j)}$ depends not only on the location of each joint but also on which nozzle is used to apply the sealant on each joint and other factors.

The goal is to identify collision-free trajectories such that all seams get processed within the minimum time span.

Generalization of the TSP

Encoding and decoding: Robot motion planning
(Schuetz et al., 2022)

Encoding

- a vector X of D · nseams random keys
- X = [x(1), x(2), …, x(nseams) | y(1), y(2), …, y(nseams) | … | z(1), z(2), …, z(nseams)]

Decoder

- INPUT: *n*-vector of random keys
  - Sort the first x(1), x(2), …, x(nseams) to determine sequence of seams to be visited by robot
  - Use a threshold logic to set other configurations: e.g.

    If conf[y(i)] ∈ {1, 2, …, C}

    Then conf[y(i)] = k if y(i) ∈ ((k-1)/ C, k/C]
  - Route robot to visit seams in order determined by x(1), x(2), …, x(nseams) using configurations determined by y(1), y(2), …, y(nseams), …, z(1), z(2), …, z(nseams) and compute sum of travel times from travel time matrix.
- OUTPUT: the routine applied to the tuples' second entry

Tree of hubs location
(Pessoa et al., 2017)

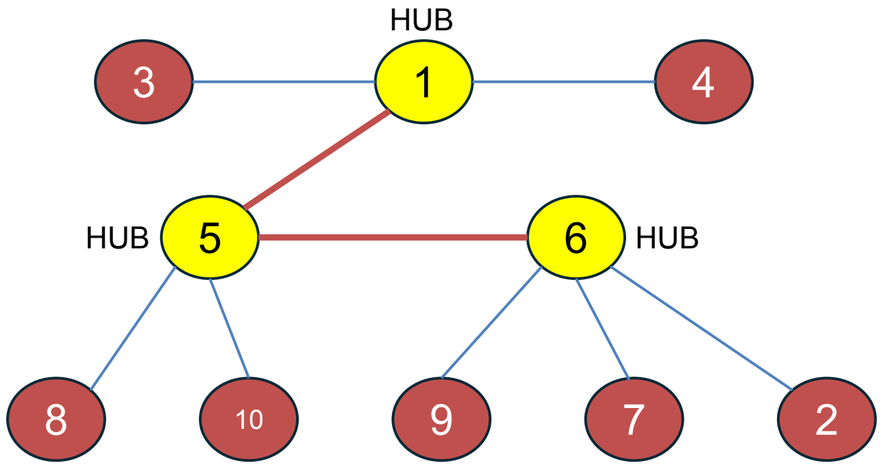
Consists of defining *p* hubs in a network that are connected by a tree with undirected edges.

From this tree, each non-hub node must be connected to a hub so that all the flow between the nodes must use the hubs to the network.

Each edge has a per-unit transportation cost, and there is a demand between each pair of nodes in the network.

The objective function to be minimized has two components:

- Transportation costs for fulfilling all demand from non-hub nodes to a designated hub
- Transportation costs of flow within the hub tree where economies of scale discount the unit cost.



Encoding and decoding: Tree of hubs location

(Pessoa et al., 2017)

Encoding

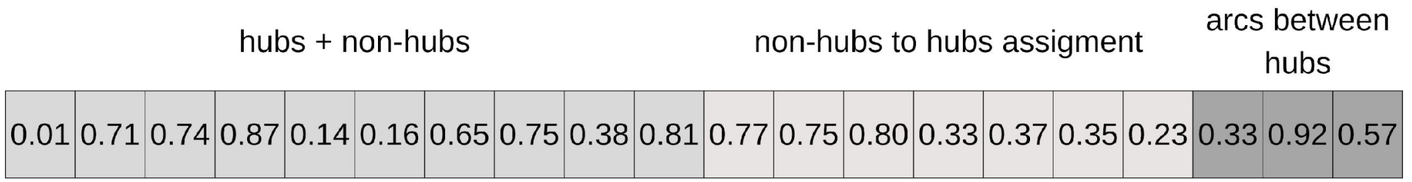
- N+(N−p) + p(p−1)/2 keys in three parts, first with N keys, second with N−p, and third with p(p−1)/2 keys, where N is the number of nodes in the network and p is the number of hubs.

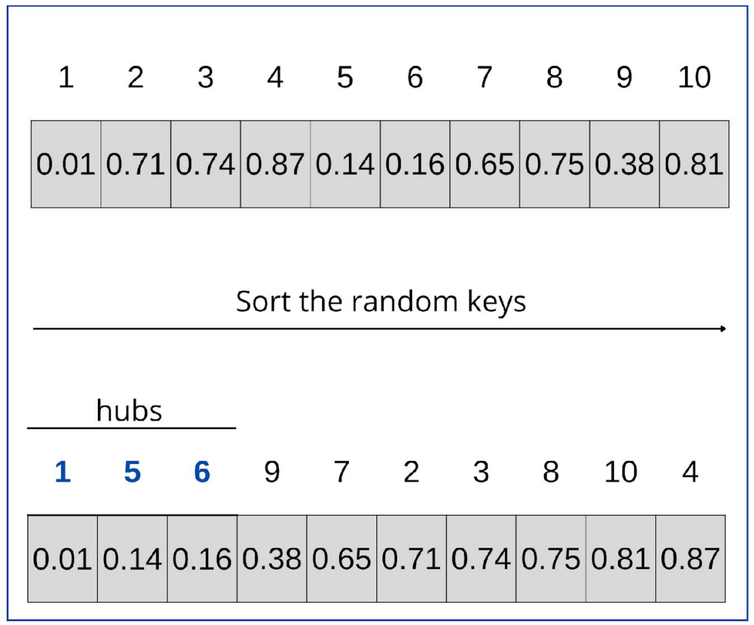
Decoder

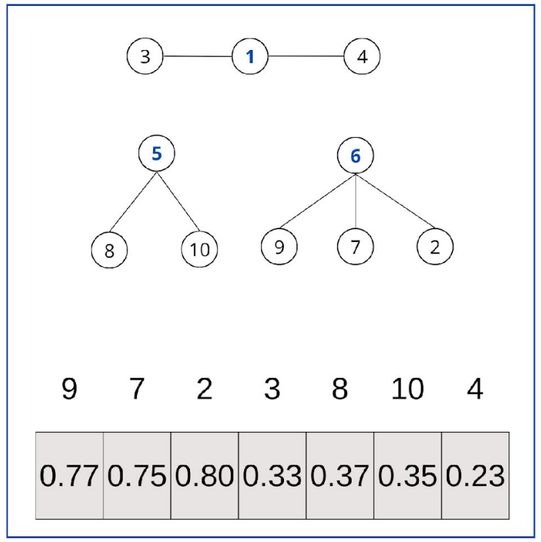
- INPUT: *n*-vector of random keys
  - Sort the first N keys and assign the indices of the p smallest keys as hubs.
  - Use the second part of the keys with thresholding to assign non-hub nodes to one of the h hubs.
  - Sort the p(p−1)/2 keys of the third part and build the inter-hub tree with a Kruskal-like algorithm, i.e. assign edges in the order of sorted keys never creating a hub cycle.
- OUTPUT: the hub location, points attribution, and tree of hubs

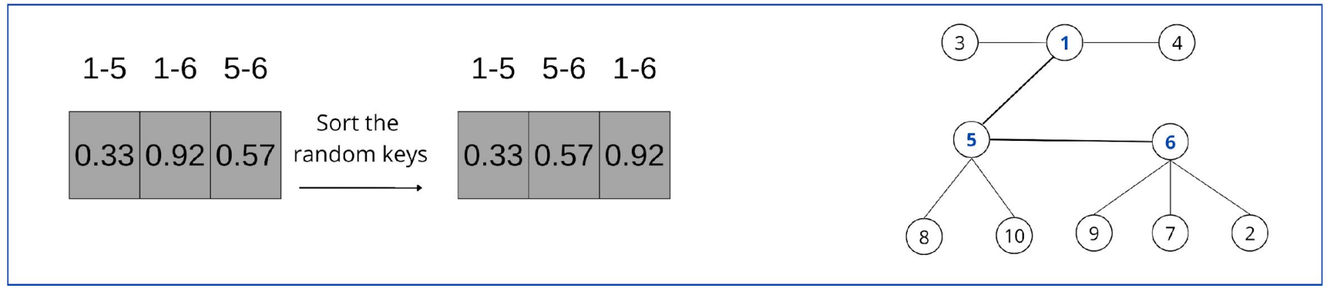
Encoding and decoding: Tree of hubs location

(Pessoa et al., 2017)









Encoding and decoding: 2D orthogonal packing

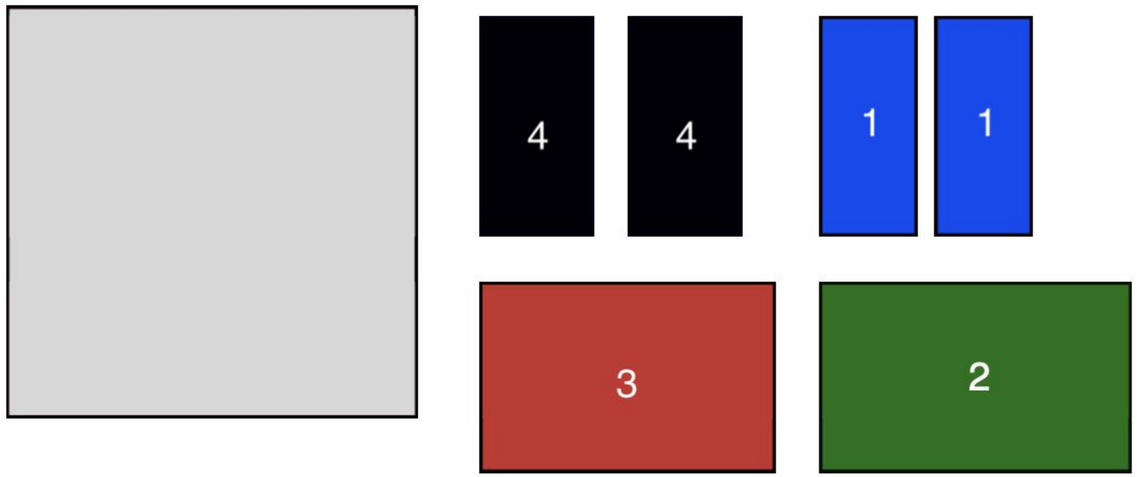
(Gonçalves and Resende, 2011b)

Encoding

- Solutions are encoded with an *n*-vector *X* of 2N random keys

Decoder

- INPUT: *n*-vector of random keys
- Sort the N keys in increasing order: Indices of sorted vector impose a sequence for placement of the N rectangles. The last N keys determine which of two placement heuristics is used to place each rectangle: Left-Bottom (LB) or Bottom-Left (BL)
- OUTPUT: a packing that maximizes the total value



Encoding and decoding: 2D orthogonal packing

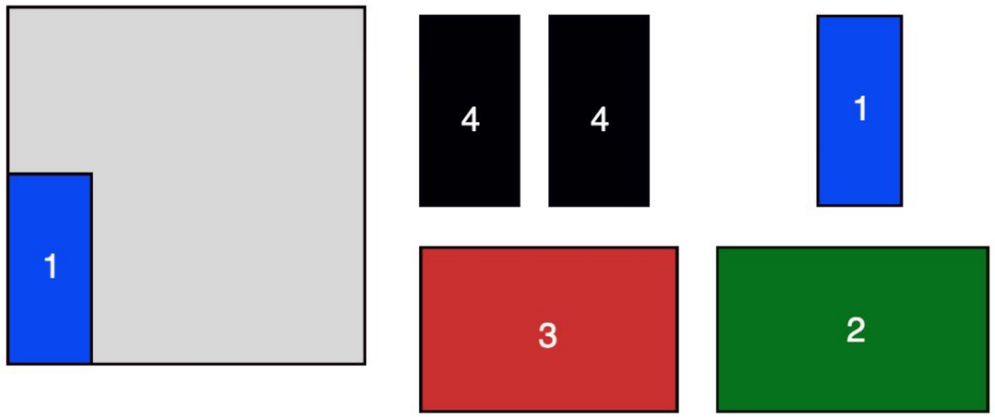
(Gonçalves and Resende, 2011b)

Encoding

X = (.12, .90, .34, .55, .88, .99 | .63, .02, .98, .21, .99, .80)

Decoder

(1, 3, 4, 5, 2, 6 | BL, LB, BL, LB, BL, BL)



Type 1 sheet placed using BL

Encoding and decoding: 2D orthogonal packing

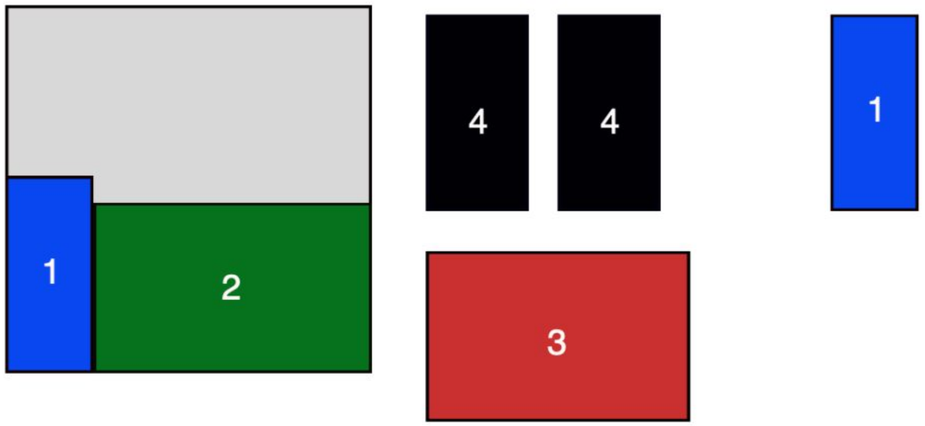
(Gonçalves and Resende, 2011b)

Encoding

X = (.12, .90, .34, .55, .88, .99 | .63, .02, .98, .21, .99, .80)

Decoder

(1, 3, 4, 5, 2, 6 | BL, LB, BL, LB, BL, BL)



Type 2 sheet placed using BL

Encoding and decoding: 2D orthogonal packing

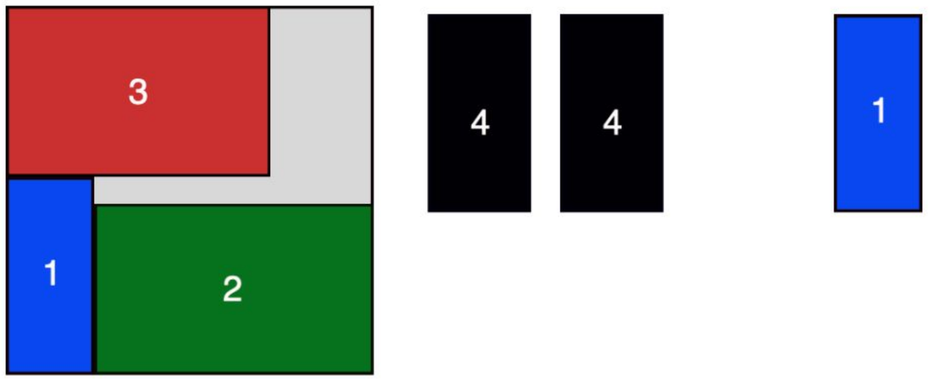
(Gonçalves and Resende, 2011b)

Encoding

X = (.12, .90, .34, .55, .88, .99 | .63, .02, .98, .21, .99, .80)

Decoder

(1, 3, 4, 5, 2, 6 | BL, LB, BL, LB, BL, BL)



Type 3 sheet placed using LB

Encoding and decoding: 2D orthogonal packing

(Gonçalves and Resende, 2011b)

Encoding

X = (.12, .90, .34, .55, .88, .99 | .63, .02, .98, .21, .99, .80)

Decoder

(1, 3, 4, 5, 2, 6 | BL, LB, BL, LB, BL, BL)

Type 4 sheet placed using BL

Remaining type 1 and type 4 sheets cannot be placed. They are left out.

Wireless backhaul network planning
(Andrade et al., 2015)

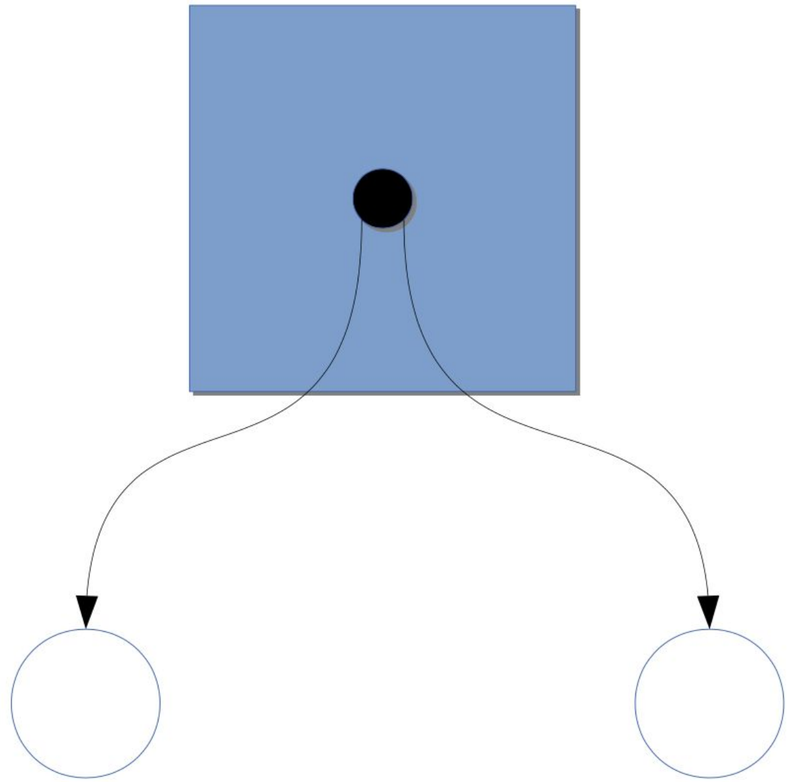
- Given a geographical region where locations are represented as lat-long coordinates, where
  - $\mathbf{V_d}$ is the set of nodes representing traffic origination points (demand points) in the region
  - $\mathbf{V_s}$ is the set of nodes representing locations where equipment for traffic collection and routing is located (e.g. utility poles)
  - $\mathbf{V_r}$ is the set of nodes representing fibered access points (FAP), e.g. remote terminals (RT), macrocell (MC), or central offices.

Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Demand splitting

- Estimate for each block total demand is placed in center of block
- Block can be served by one or more antennae so demand can be split among them
- Because of this undirected cycles can be introduced resulting in a directed acyclic graph (DAG)



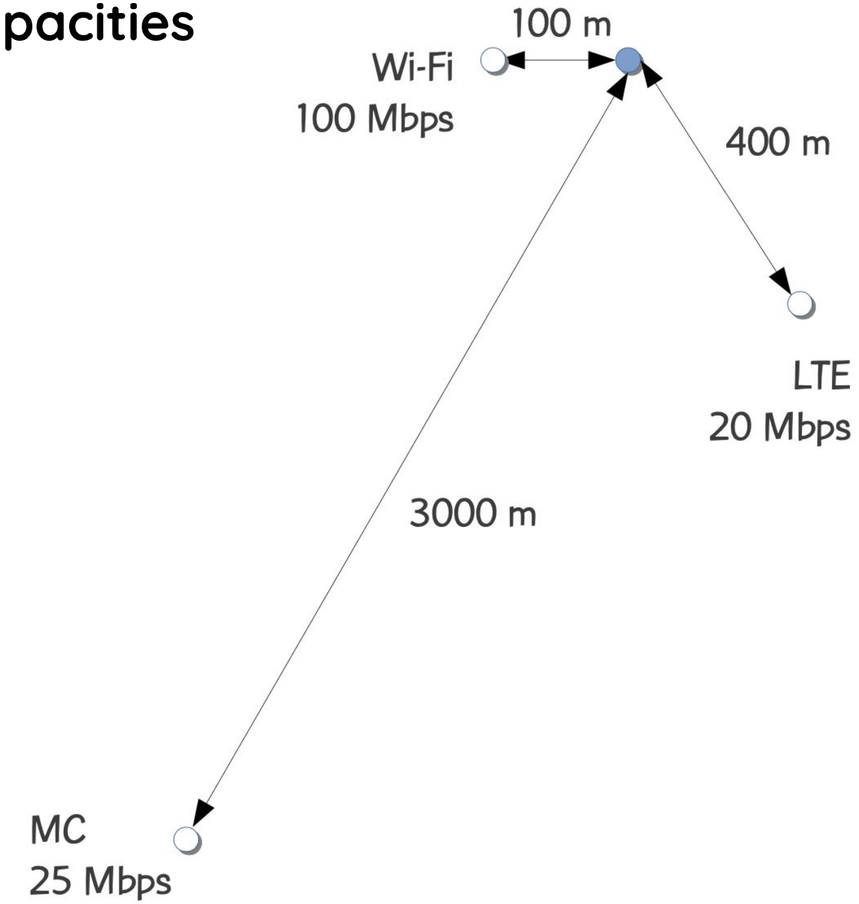
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Access equipment action radii & capacities

- Action radii
  - Wi-Fi:100m
  - LTE: 400 m
  - macrocell: 3 km

- Processing capacity
  - Wi-Fi: 100 Mbps
  - LTE: 20 Mbps
  - macrocell: 25 Mbps



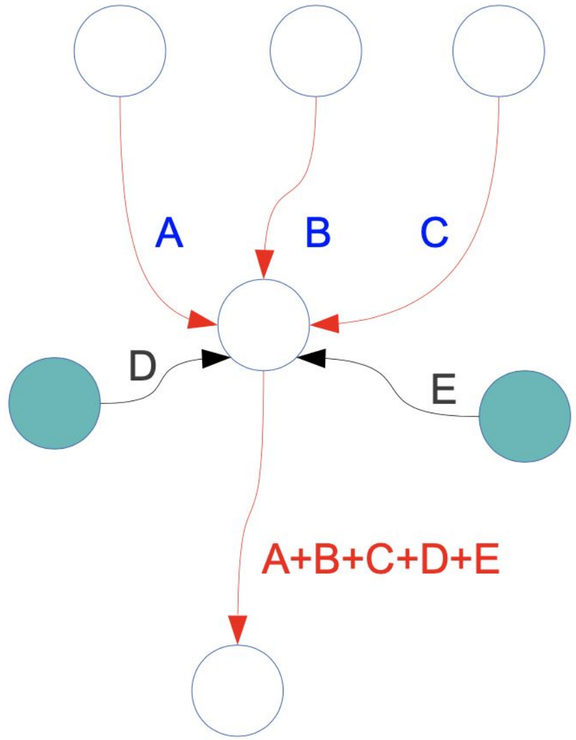
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Retransmitter equipment capacity

- Action radii:
  - 1 km

- Processing capacity
  - Flow that equipment receives from other wireless retransmitters plus flow it sends to other retransmitters is limited to at most 100 Mbps



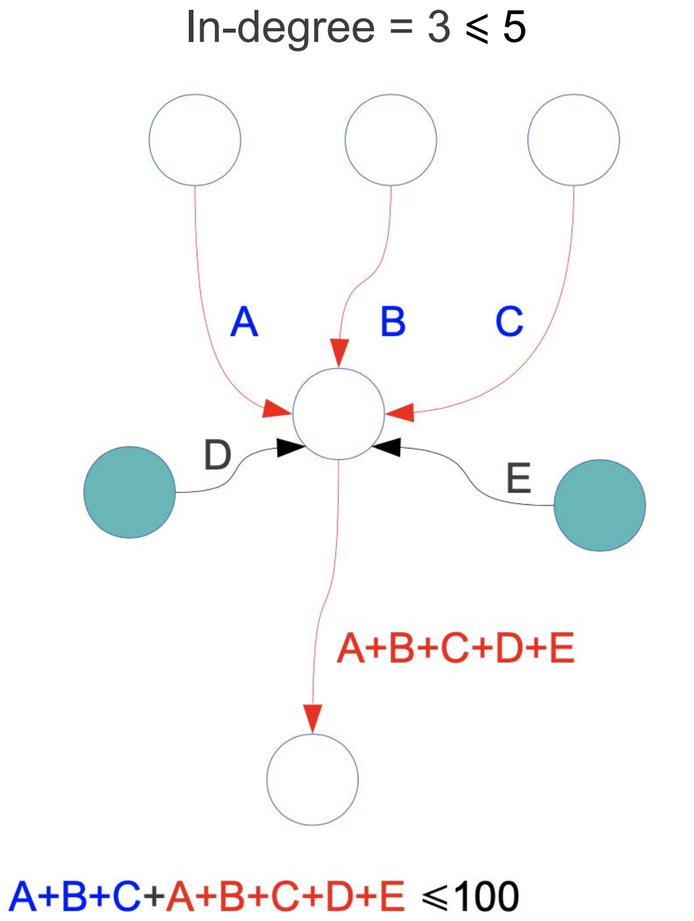
$A+B+C+A+B+C+D+E \leqslant 100$

Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Retransmitter equipment capacity

- Fan-in constraint
  - A limited number (5) of neighboring transmission equipment can flow traffic into equipment

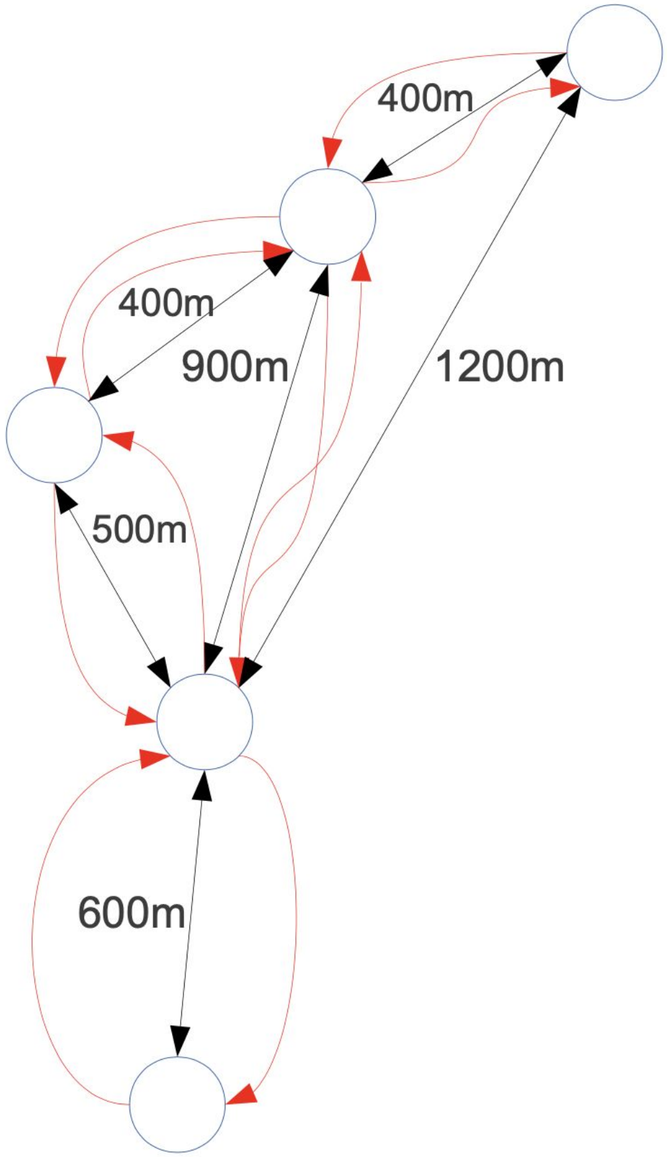


Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Line of sight and minimum distance

- PCSF Problem assumed, for each pair u,v ∈Vs that (u,v) ∈ E and (v,u) ∈ E
- Not so in wireless backhaul planning: (u,v) ∈ E and (v,u) ∈ E only if
  - Poles u and v are within 1000 m of each other

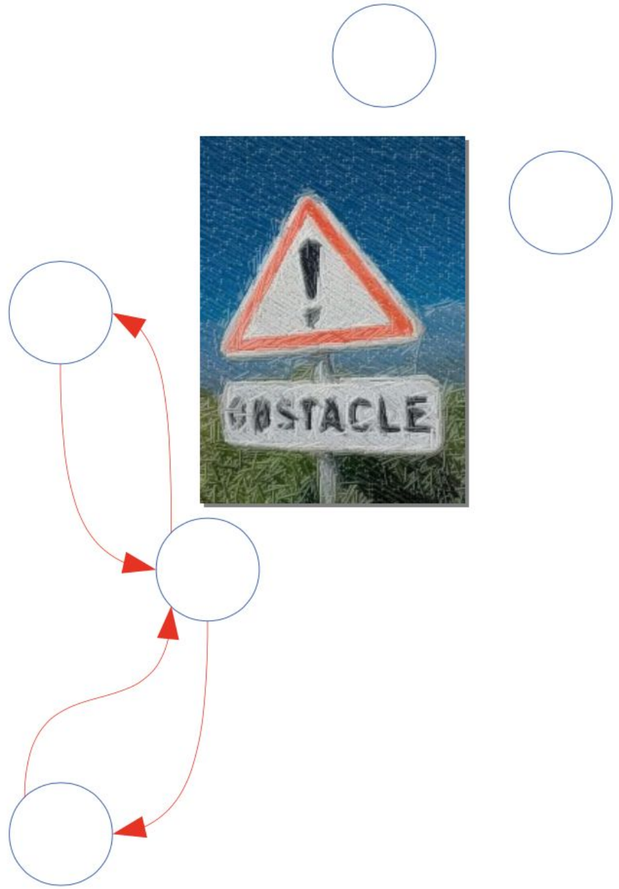


Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Line of sight and minimum distance

- PCSF Problem assumed, for each pair u,v ∈Vs that (u,v) ∈ E and (v,u) ∈ E
- Not so in wireless backhaul planning: (u,v) ∈ E and (v,u) ∈ E only if
  - Poles u and v are within 1000 m of each other
  - Poles u and v are in each other's line of sight

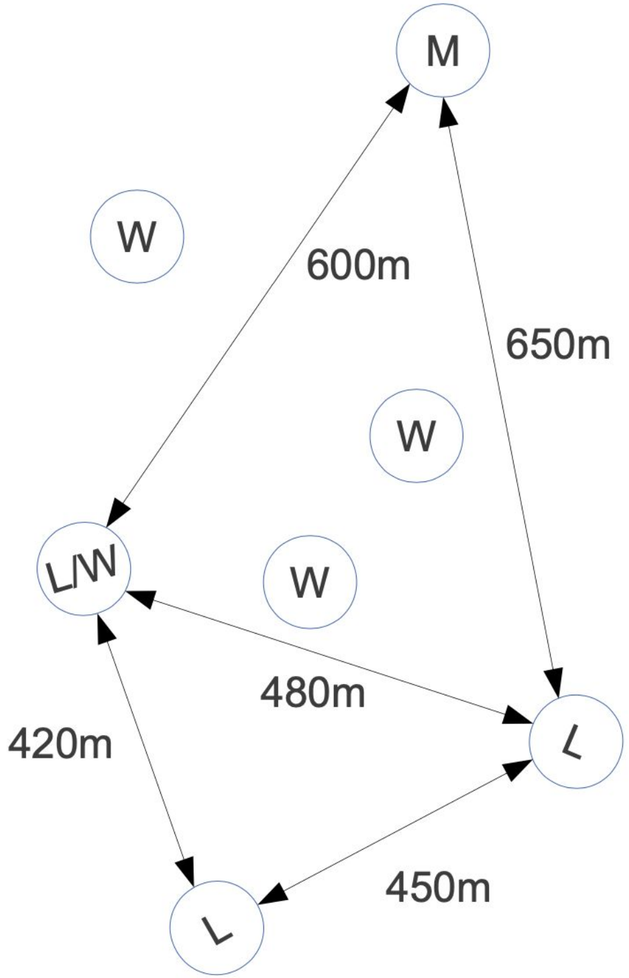


Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Interference

- LTE and MC use licensed spectrum and can interfere with each other.
  - Pairs of LTE antennae must be separated by at least 400m
  - LTE and macrocells by at least 500m
  - No constraint on Wi-Fi exists since it uses non-licensed spectrum

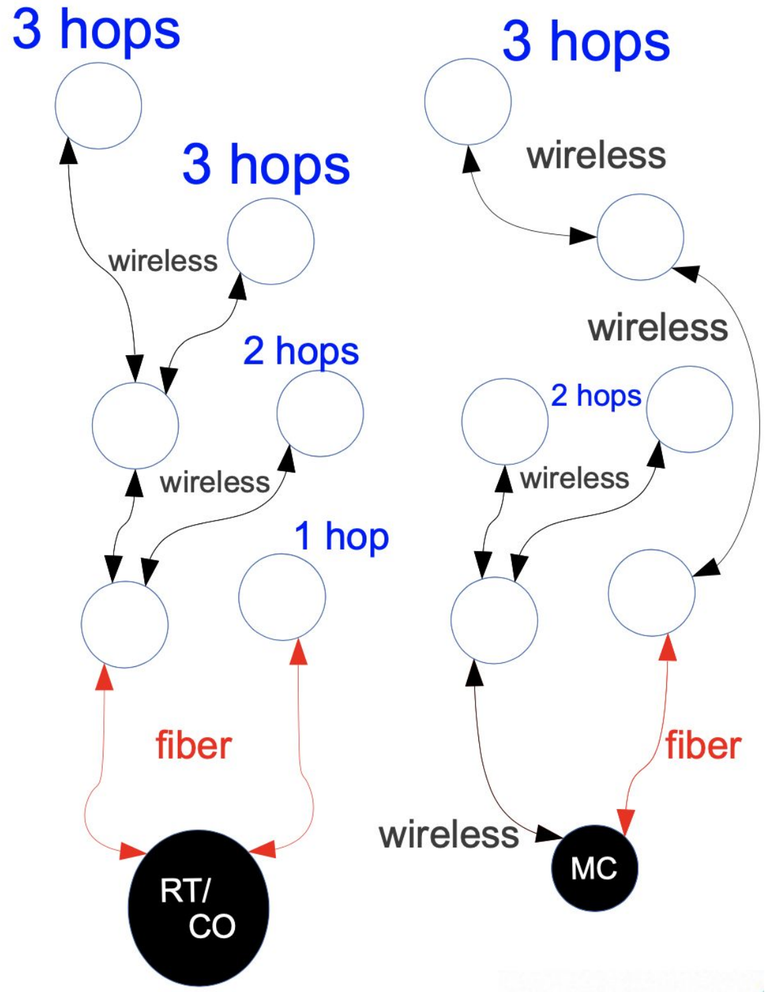


Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: k-Hops

- First hop is from FAP (root) to pole
  - If root is Central Office (CO) or RT link must use fiber
  - If root is macrocell link can be fiber or wireless
- All other links are wireless
- Number of hops is limited to k = 2 or 3 (in case first link is fiber)



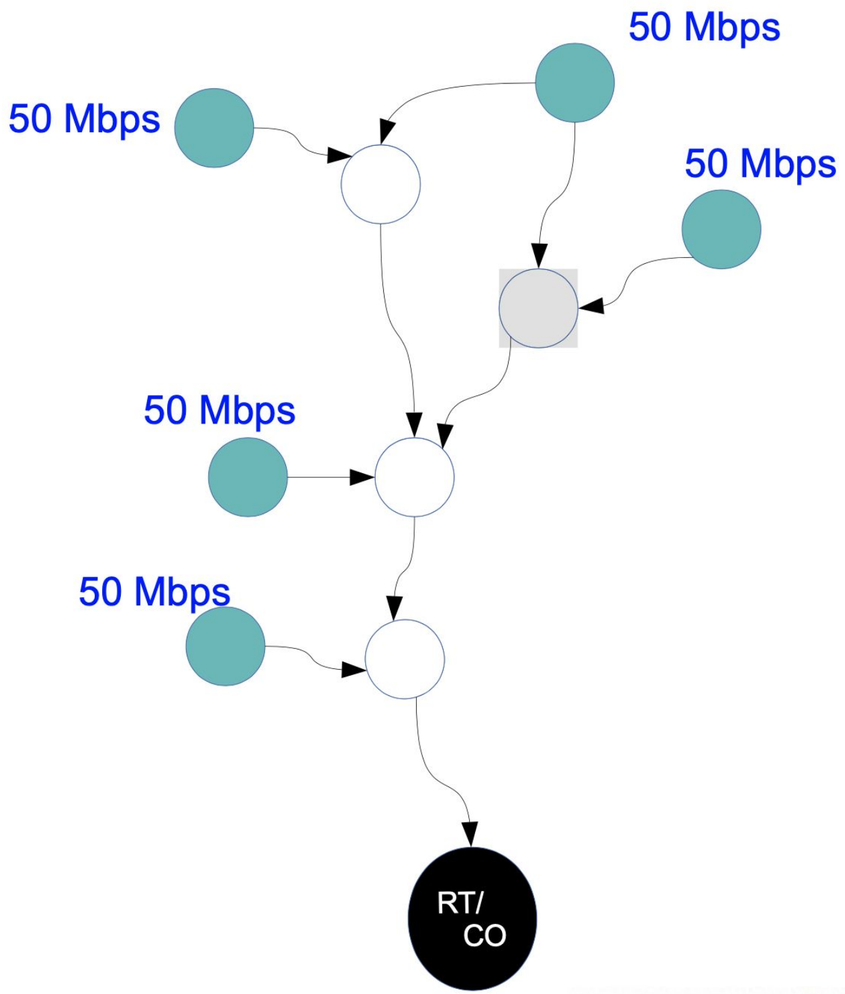
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Traffic flow

- Traffic that is backhauled from demand points to root nodes of forest is limited by equipment capacities.

- Only traffic that reaches roots is counted as revenue.



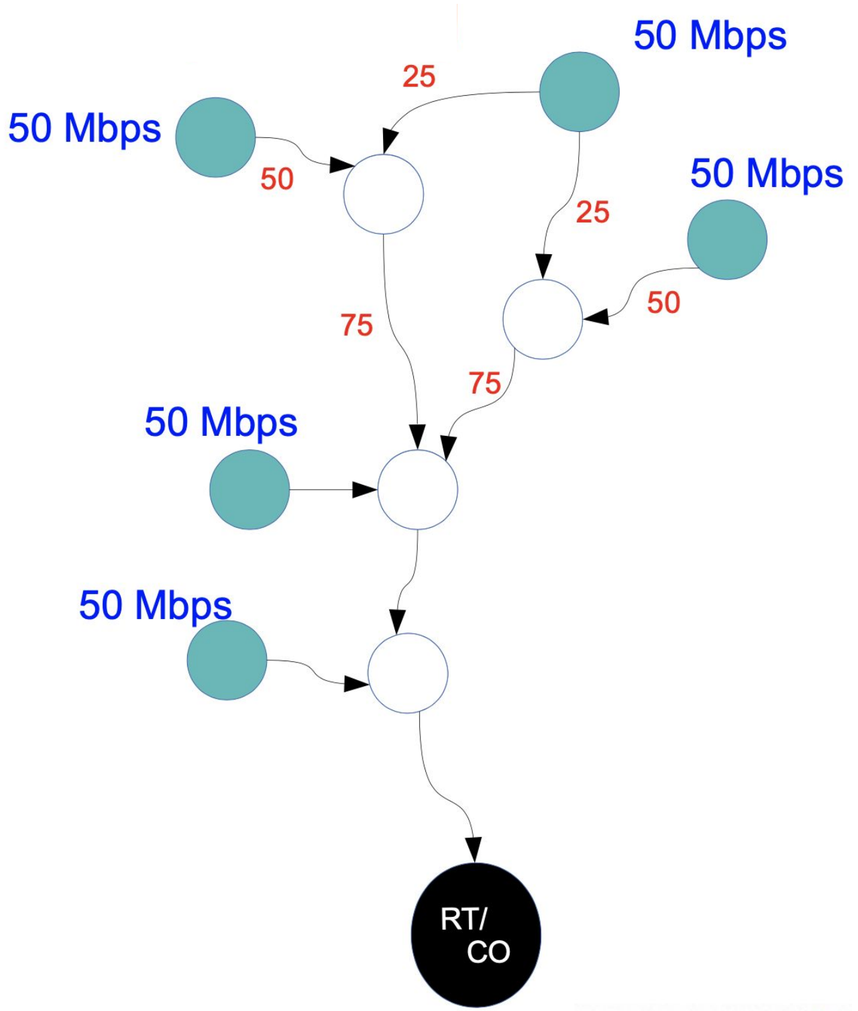
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Traffic flow

- Traffic that is backhauled from demand points to root nodes of forest is limited by equipment capacities.

- Only traffic that reaches roots is counted as revenue.



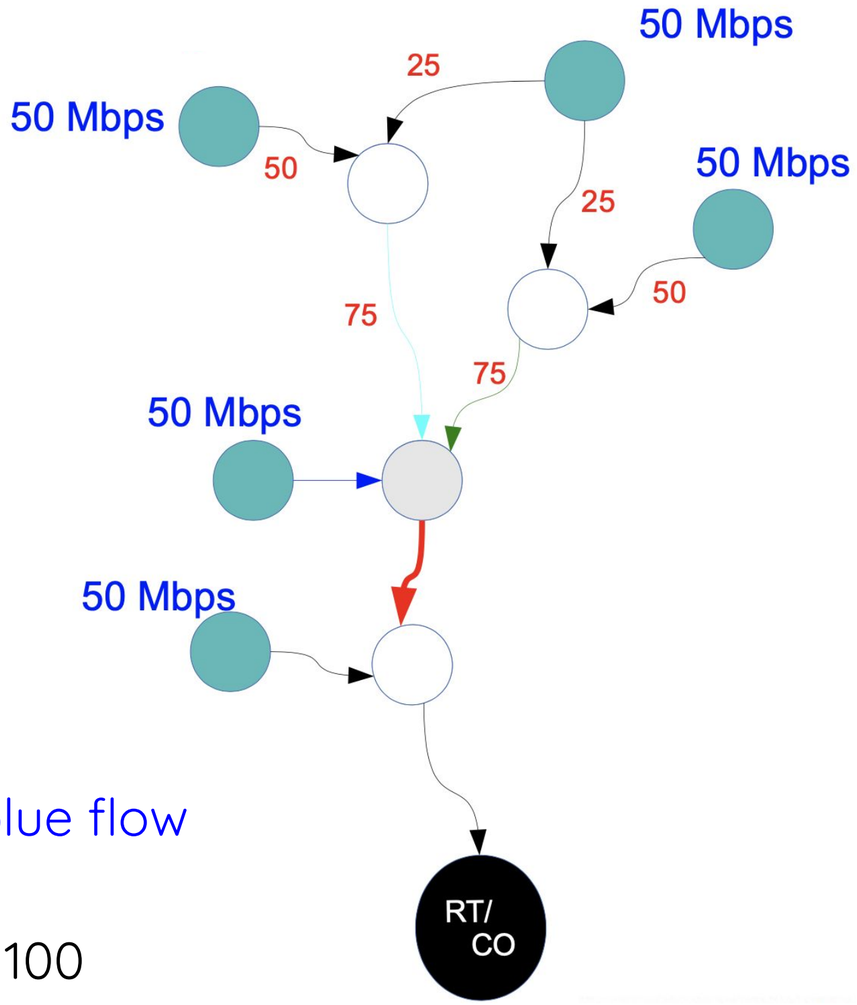
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Traffic flow

- Traffic that is backhauled from demand points to root nodes of forest is limited by equipment capacities.

- Only traffic that reaches roots is counted as revenue.



Flow conservation: red flow = cyan flow + green flow + blue flow

Capacity constraint: red flow + cyan flow + green flow ≤ 100

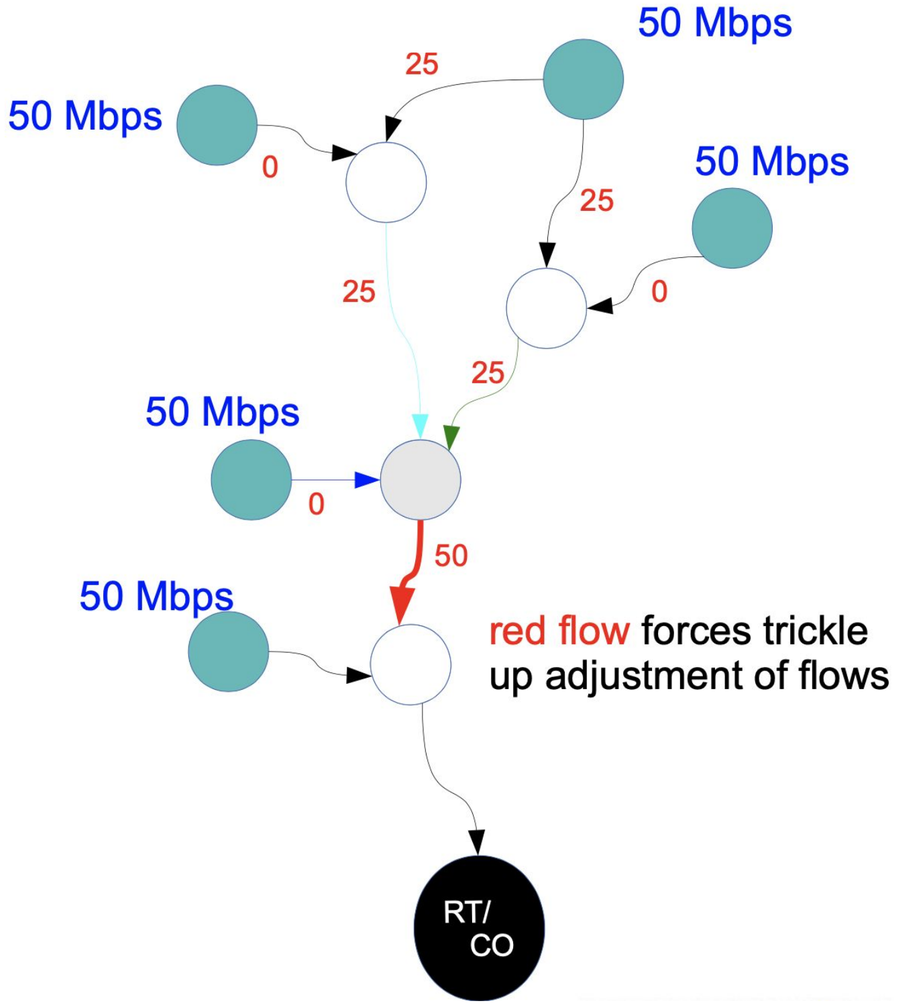
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Traffic flow

- Traffic that is backhauled from demand points to root nodes of forest is limited by equipment capacities.

- Only traffic that reaches roots is counted as revenue.



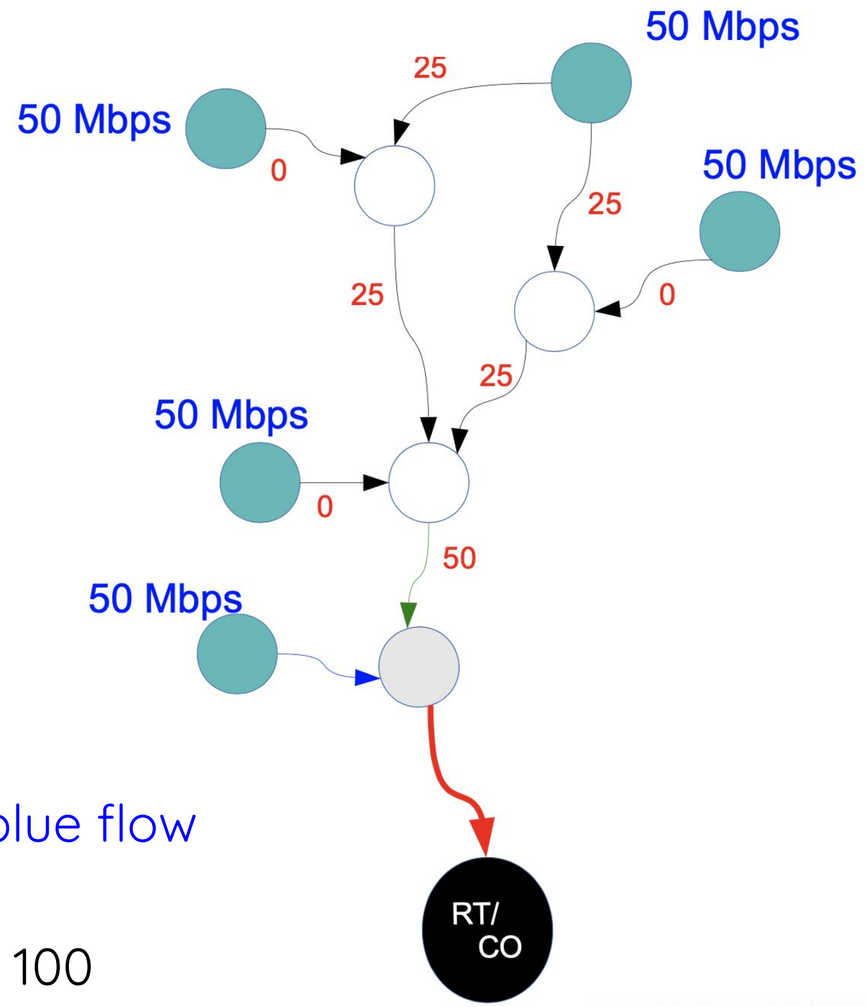
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Traffic flow

- Traffic that is backhauled from demand points to root nodes of forest is limited by equipment capacities.

- Only traffic that reaches roots is counted as revenue.



Flow conservation: red flow = cyan flow + green flow + blue flow

Capacity constraint: red flow + cyan flow + green flow ≤ 100

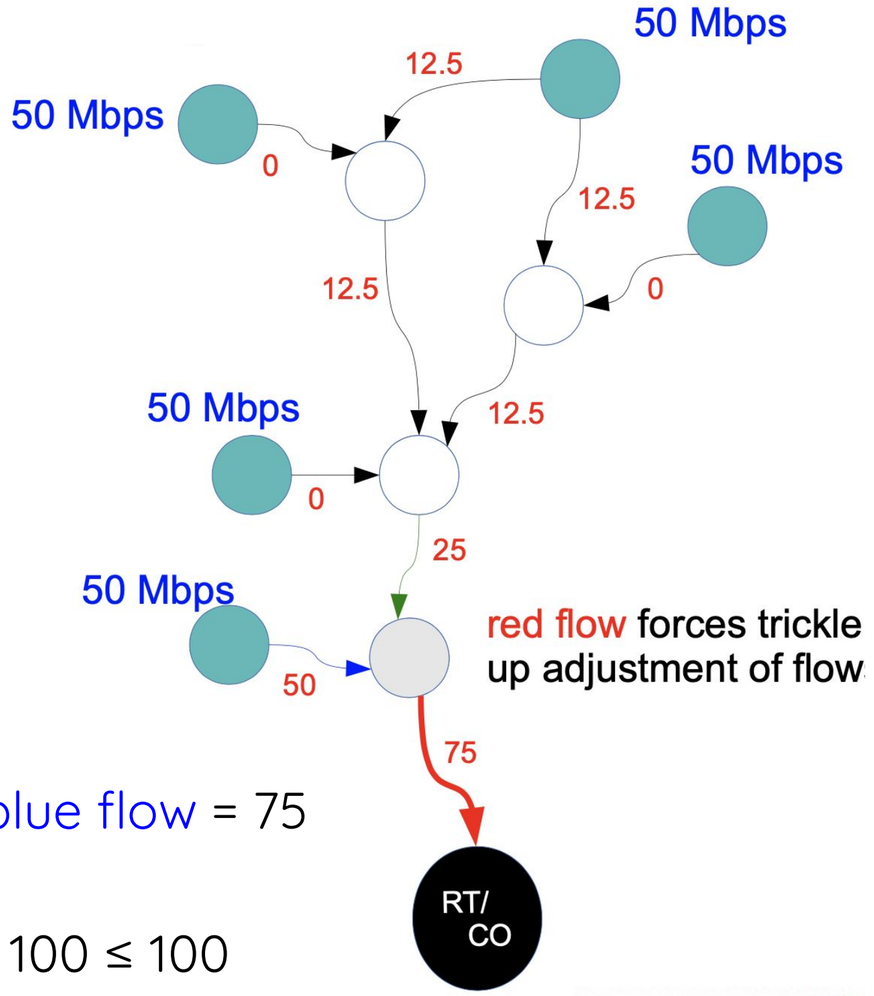
Wireless backhaul network planning

(Andrade et al., 2015)

Constraints: Traffic flow

- Traffic that is backhauled from demand points to root nodes of forest is limited by equipment capacities.

- Only traffic that reaches roots is counted as revenue.



Flow conservation: red flow = cyan flow + green flow + blue flow = 75

Capacity constraint: red flow + cyan flow + green flow = 100 ≤ 100

Wireless backhaul network planning

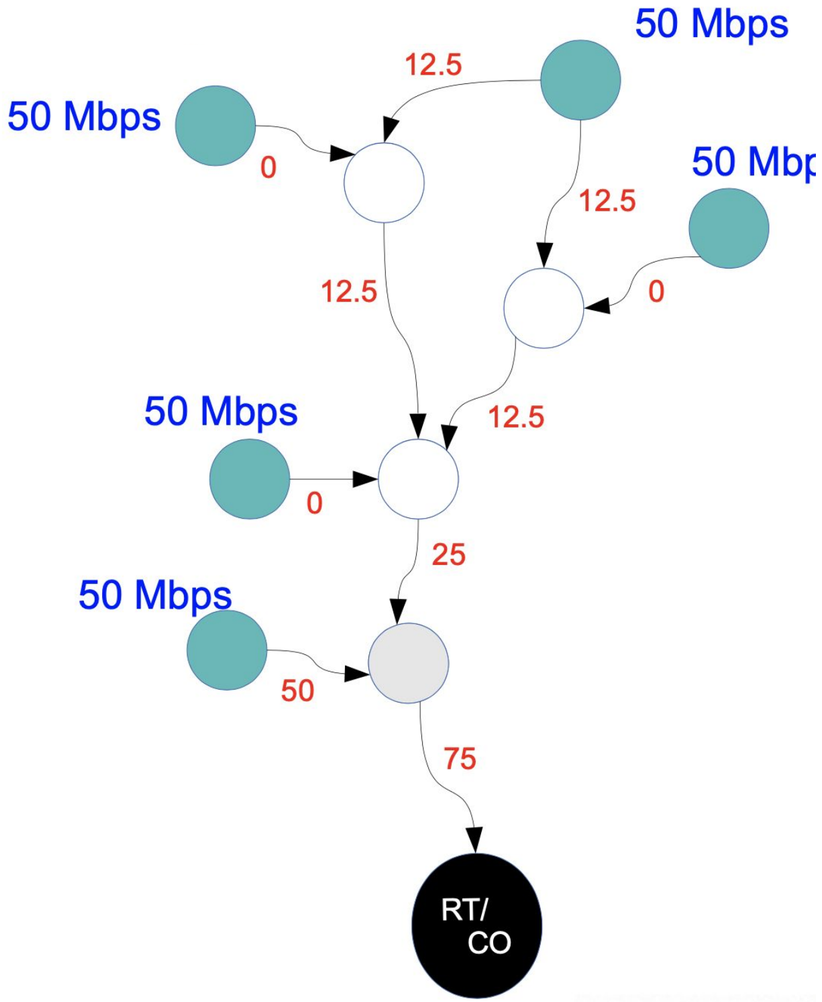
(Andrade et al., 2015)

Constraints: Traffic flow

- Traffic that is backhauled from demand points to root nodes of forest is limited by equipment capacities.

- Only traffic that reaches roots is counted as revenue.

Of the 250 Mbps available traffic, only 75 Mbps reaches root node.



Wireless backhaul network planning

(Andrade et al., 2015)

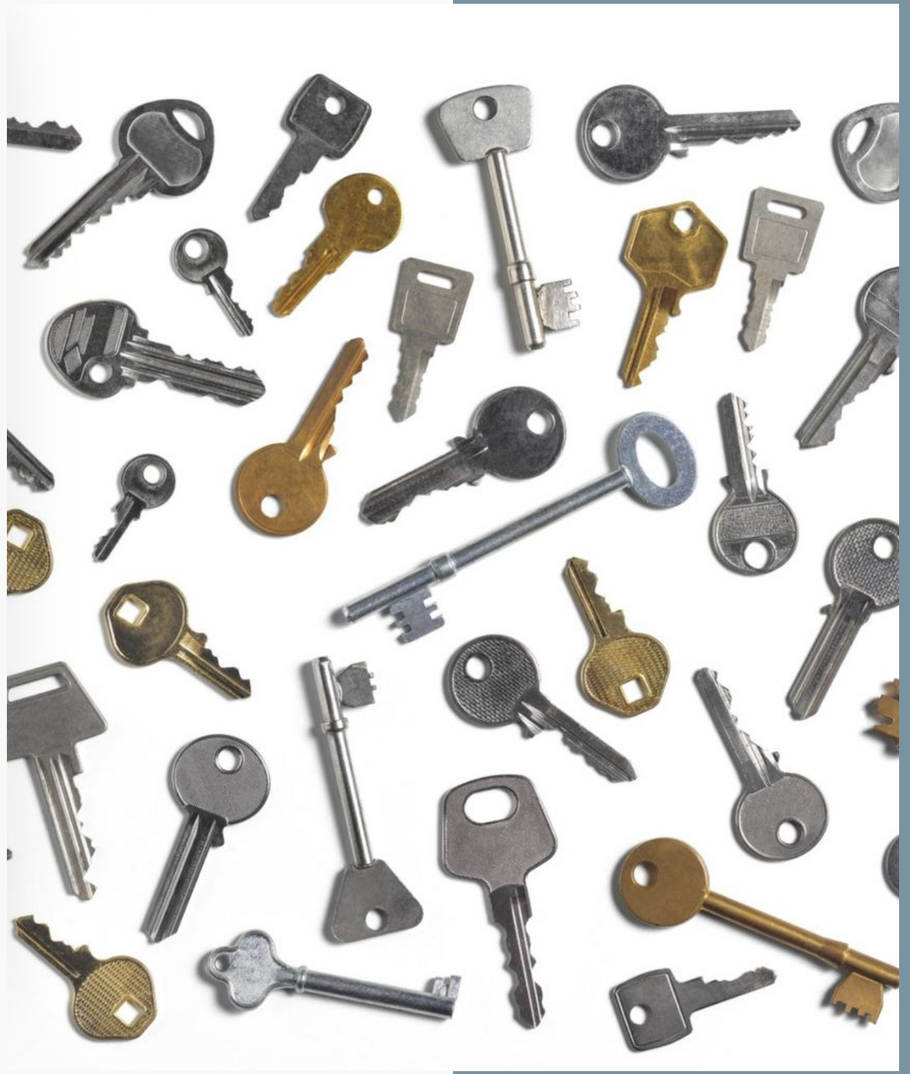
Encoding

- Solutions are encoded with an *n*-vector *X* of random keys, where n = 5 x # of poles

Decoder

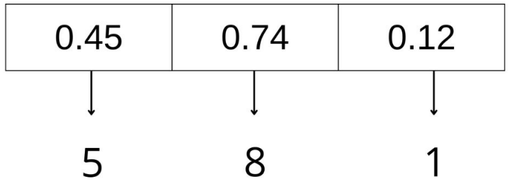
- Define activation order & install LTE on poles
- Build backhaul graph
- Remove unused equipment
- Compute maximum flow from demand points to FAPs Remove unused equipment & poles
- Compute cost and revenue and return objective function value

RKO applications in discrete optimization problems



α-Neighbor p-Median Problem

Encoding



Open facilities:

Decoder

k	0	1	2	3	4	5	6	7	8	9
C = [	1	2	3	4	**5**	6	7	8	9	10]

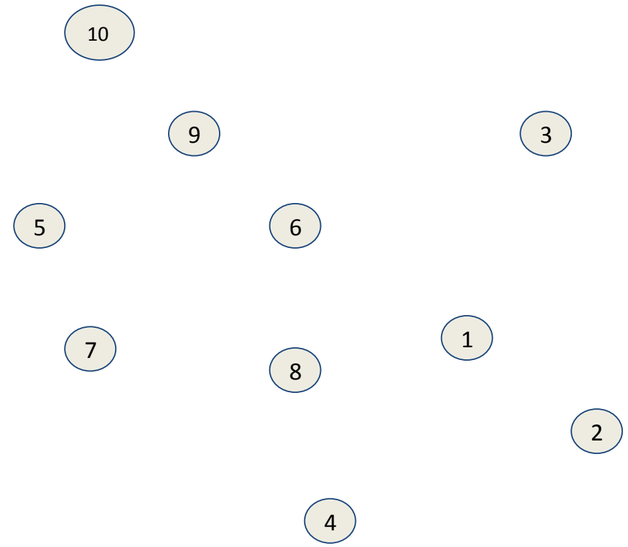
$k = \lfloor 0.45 \times 10 \rfloor = 4$ — facility 5

k	0	1	2	3	4	5	6	7	8
C = [	1	2	3	4	6	7	**8**	9	10]

$k = \lfloor 0.74 \times 9 \rfloor = 6$ — facility 8

k	0	1	2	3	4	5	6	7
C = [	1	2	3	4	6	7	9	10]

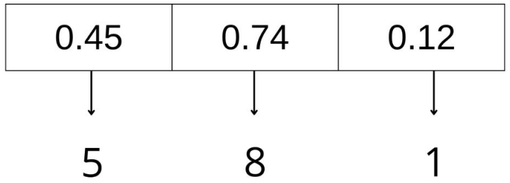
$k = \lfloor 0.12 \times 8 \rfloor = 0$ — facility 1



α = 2, p = 3, n = 10

α-Neighbor p-Median Problem

Encoding



Open facilities:

Decoder

k	0	1	2	3	4	5	6	7	8	9
C = [	1	2	3	4	**5**	6	7	8	9	10]

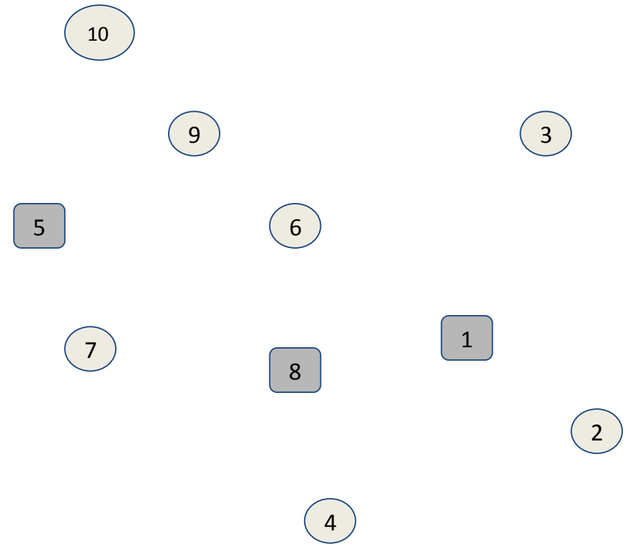
k = ⌊0.45 × 10⌋= 4 — facility 5

k	0	1	2	3	4	5	6	7	8
C = [	1	2	3	4	6	7	**8**	9	10]

k = ⌊0.74 × 9⌋= 6 — facility 8

k	0	1	2	3	4	5	6	7
C = [	1	2	3	4	6	7	9	10]

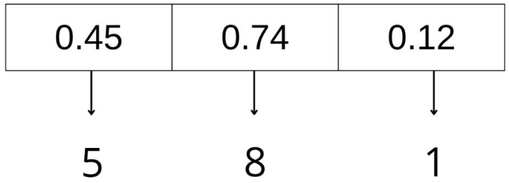
k = ⌊0.12 × 8⌋= 0 — facility 1



α = 2, p = 3, n = 10

α-Neighbor p-Median Problem

Encoding



Open facilities:

Decoder

k	0	1	2	3	4	5	6	7	8	9
C = [	1	2	3	4	**5**	6	7	8	9	10]

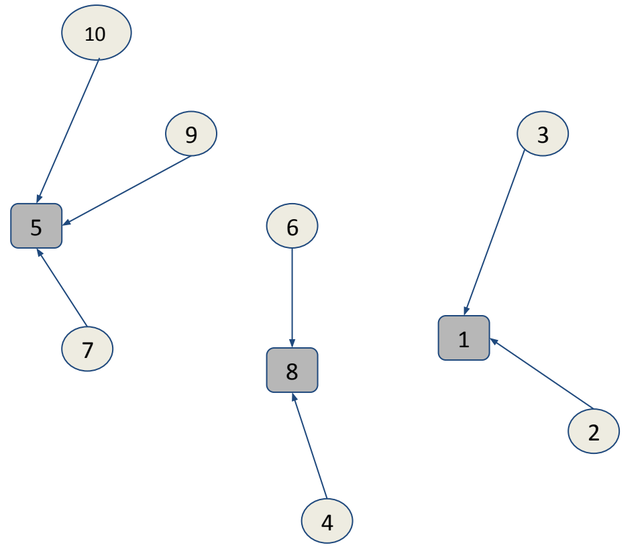
k = ⌊0.45 × 10⌋= 4 — facility 5

k	0	1	2	3	4	5	6	7	8
C = [	1	2	3	4	6	7	**8**	9	10]

k = ⌊0.74 × 9⌋= 6 — facility 8

k	0	1	2	3	4	5	6	7
C = [	1	2	3	4	6	7	9	10]

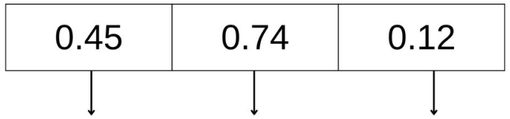
k = ⌊0.12 × 8⌋= 0 — facility 1



α = 2, p = 3, n = 10

α-Neighbor p-Median Problem

Encoding



Open facilities:

Decoder

k	0	1	2	3	4	5	6	7	8	9
C = [	1	2	3	4	**5**	6	7	8	9	10]

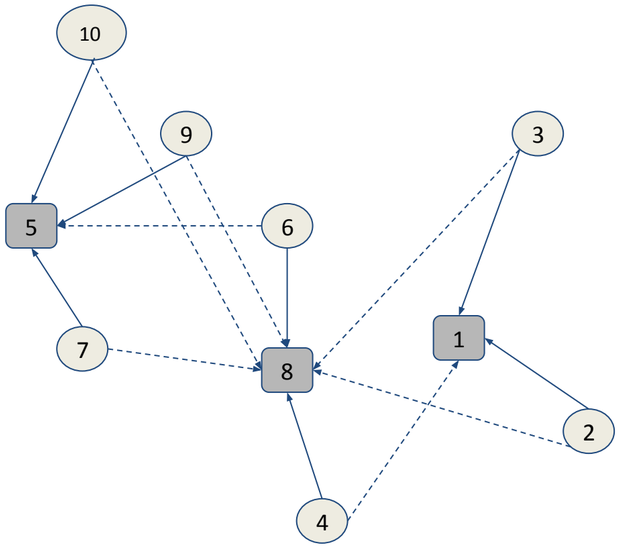
$k = \lfloor 0.45 \times 10 \rfloor = 4$ — facility 5

k	0	1	2	3	4	5	6	7	8
C = [	1	2	3	4	6	7	**8**	9	10]

$k = \lfloor 0.74 \times 9 \rfloor = 6$ — facility 8

k	0	1	2	3	4	5	6	7
C = [	1	2	3	4	6	7	9	10]

$k = \lfloor 0.12 \times 8 \rfloor = 0$ — facility 1



α = 2, p = 3, n = 10

α-Neighbor p-Median Problem

Parameters of the RKO metaheuristics to solve the αNpMP.

Parameters	Definition	BRKGA	GA	SA	ILS	VNS	GRASP	PSO	LNS
p	population size	1597	1000					100	
p_e	elite set	0.10							
p_m	mutant set	0.20							
ρ	inherit probability	0.70							
p_c	crossover probability		0.85						
μ	mutation probability		0.03						
T_0	initial temperature			10000					1000
SA_{max}	number of iterations			100					
α	cooling rate			0.99					0.90
β_{min}	minimum rate of shaking			0.10	0.15	0.05			0.10
β_{max}	maximum rate of shaking			0.20	0.40				0.30
k_{max}	number of neighborhoods					6			
h_s	start grid dense						0.125		
h_e	end grid dense						0.00012		
c_1	cognitive coefficient							2.05	
c_2	social coefficient							2.05	
w	inertia weight							0.73	

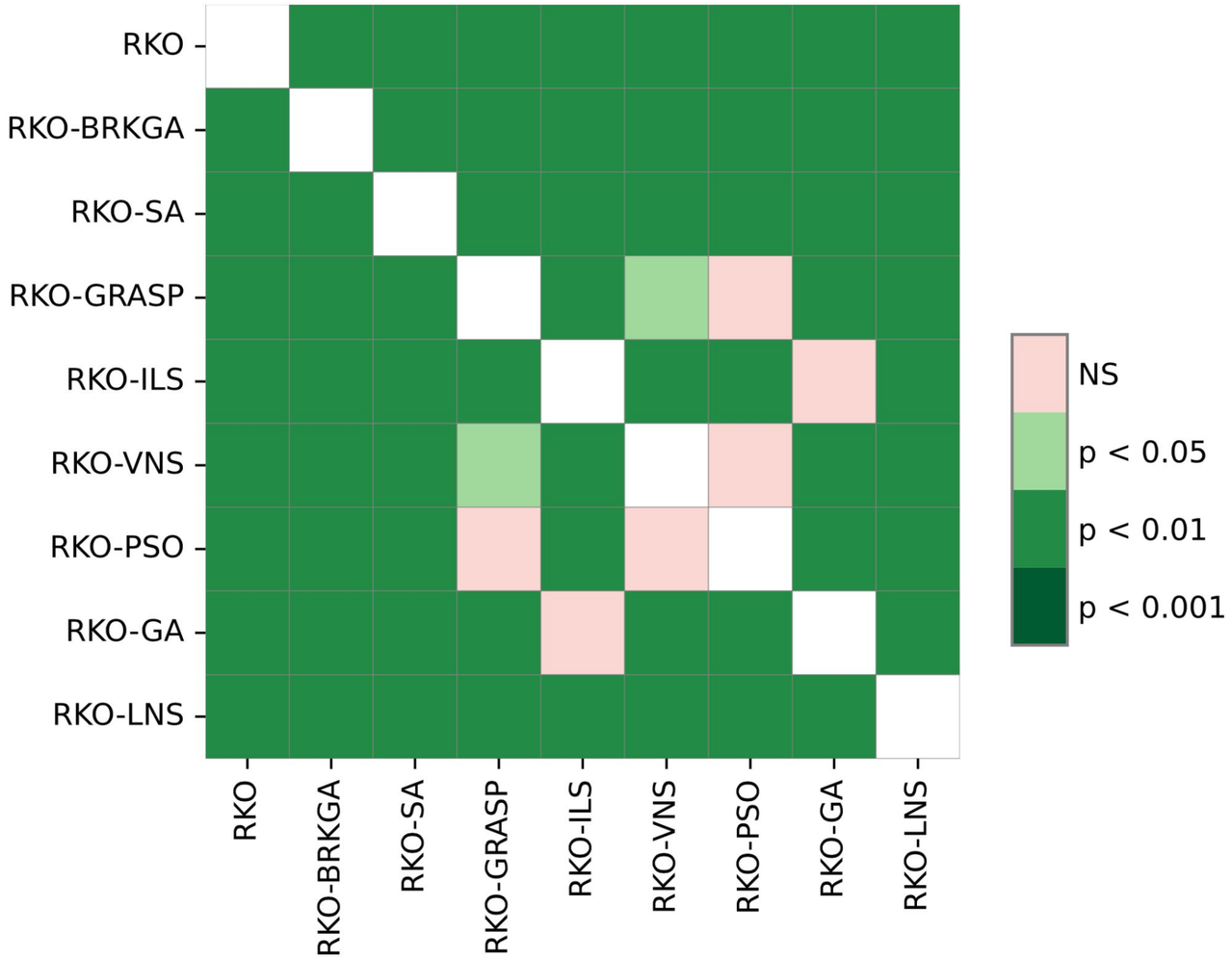
α-Neighbor p-Median Problem

Summary of the αNpMP results for 80 instances of the OR-library.

Method	$Best$	RPD_{best}	RPD_{aver}	best found at (s)	$\#BKS$
Gurobi[2]	84296.53	0.00	-	155.67	**80**
BiMM[1]	86465.14	2.55	-	2.43	0
BP-VNS[2]	84296.53	0.00	-	0.38	**80**
RKO	84300.39	0.003	0.02	28.70	**65**
RKO-BRKGA	84310.84	0.01	0.07	23.11	54
RKO-SA	84349.73	0.04	0.10	38.67	51
RKO-GRASP	84328.86	0.03	0.10	41.02	51
RKO-ILS	84321.18	0.02	0.06	44.73	53
RKO-VNS	84321.68	0.02	0.06	42.43	55
RKO-PSO	84356.23	0.05	0.16	35.34	39
RKO-GA	84312.56	0.01	0.04	33.74	55
RKO-LNS	84335.30	0.03	0.12	38.84	50

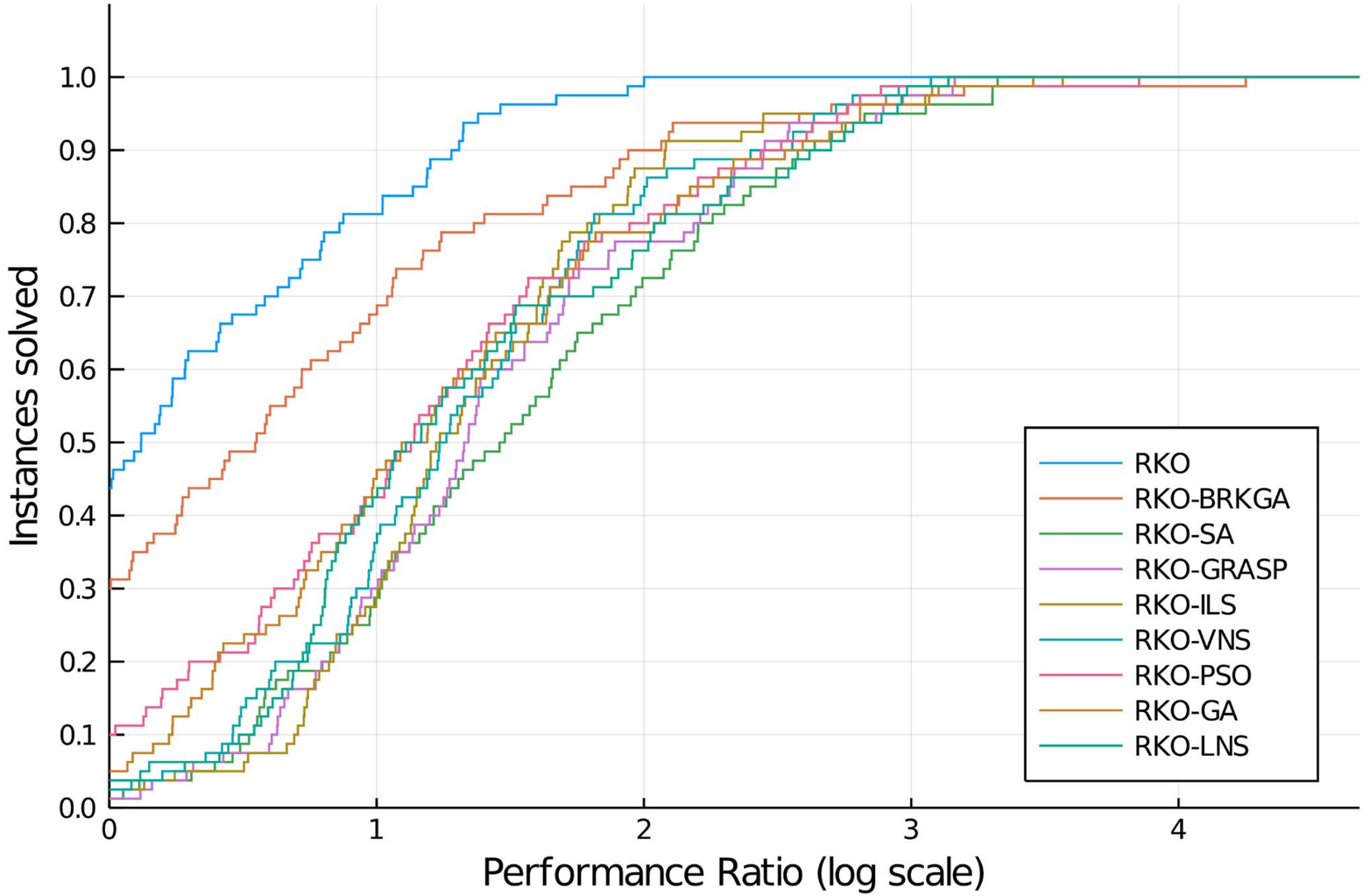
[1] (Panteli et al., 2021) [2] (Chagas et al., 2024)

α-Neighbor p-Median Problem



Heat map of the results (p-values) of the Nemenyi test for the RKO methods to solve the αNpMP.

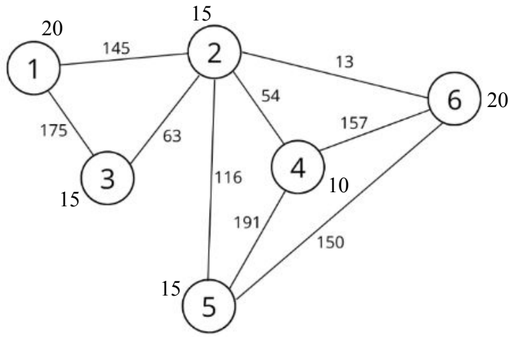
α-Neighbor p-Median Problem



Performance profile of runtime for RKO methods to solve the αNpMP.

Node Capacitated Graph Partitioning Problem

|B| = 6, |N| = 2, C = 50



Encoding

1	2	3	4	5	6	#N
0.8	0.1	0.5	0.2	0.4	0.6	0.7

Sort the random keys →

2	4	5	3	6	1	#N
0.1	0.2	0.4	0.5	0.6	0.8	0.7

RNC 1
RNC 2

Decoder

Node Capacitated Graph Partitioning Problem

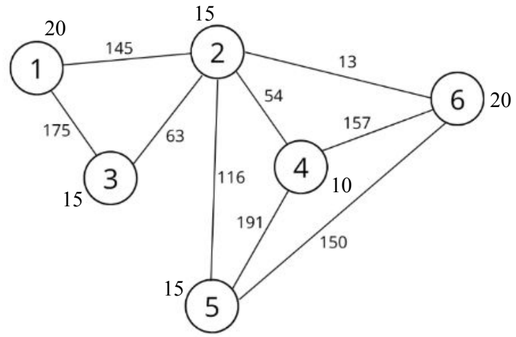
|B| = 6, |N| = 2, C = 50

Encoding

1	2	3	4	5	6	#N
0.8	0.1	0.5	0.2	0.4	0.6	0.7

Sort the random keys

2	4	5	3	6	1	#N
0.1	0.2	0.4	0.5	0.6	0.8	0.7



RNC 1	2
RNC 2	4

Decoder

Node Capacitated Graph Partitioning Problem

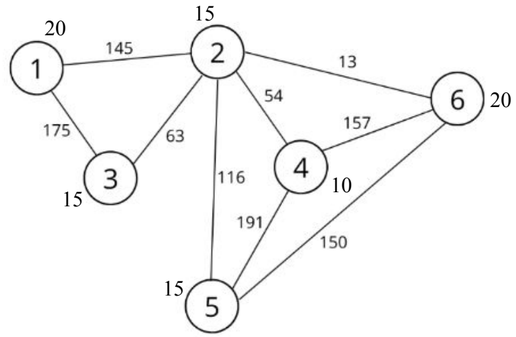
|B| = 6, |N| = 2, C = 50

Encoding

1	2	3	4	5	6	#N
0.8	0.1	0.5	0.2	0.4	0.6	0.7

Sort the random keys

2	4	5	3	6	1	#N
0.1	0.2	0.4	0.5	0.6	0.8	0.7



RNC 1	2	
RNC 2	4	5

Decoder

Node Capacitated Graph Partitioning Problem

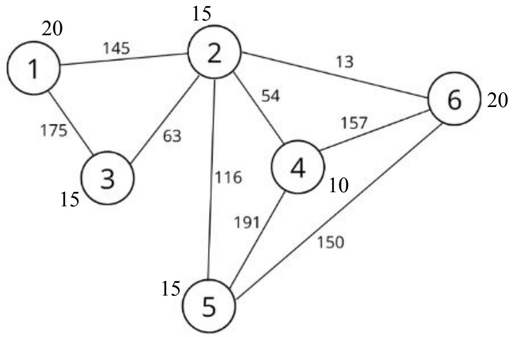
|B| = 6, |N| = 2, C = 50

Encoding

1	2	3	4	5	6	#N
0.8	0.1	0.5	0.2	0.4	0.6	0.7

Sort the random keys

2	4	5	3	6	1	#N
0.1	0.2	0.4	0.5	0.6	0.8	0.7



RNC 1	2	3
RNC 2	4	5

Decoder

Node Capacitated Graph Partitioning Problem

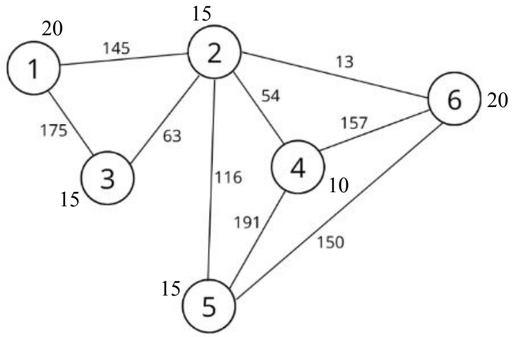
|B| = 6, |N| = 2, C = 50

Encoding

1	2	3	4	5	6	#N
0.8	0.1	0.5	0.2	0.4	0.6	0.7

Sort the random keys

2	4	5	3	6	1	#N
0.1	0.2	0.4	0.5	0.6	0.8	0.7

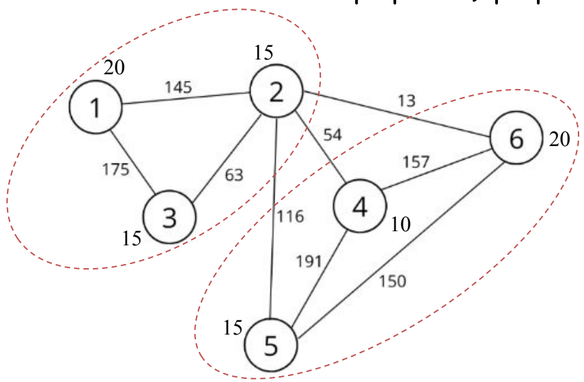


RNC 1	2	3	
RNC 2	4	5	6

Decoder

Node Capacitated Graph Partitioning Problem

|B| = 6, |N| = 2, C = 50



Encoding

1	2	3	4	5	6	#N
0.8	0.1	0.5	0.2	0.4	0.6	0.7

Sort the random keys →

2	4	5	3	6	1	#N
0.1	0.2	0.4	0.5	0.6	0.8	0.7

RNC 1	2	3	1
RNC 2	4	5	6

Decoder

Node Capacitated Graph Partitioning Problem

Parameters of the RKO metaheuristics to solve the αNpMP.

Parameters	Definition	BRKGA	GA	SA	ILS	VNS	GRASP	PSO	LNS
p	population size	1597	1000					50	
p_e	elite set	0.10							
p_m	mutant set	0.20							
ρ	inherit probability	0.70							
p_c	crossover probability		0.85						
μ	mutation probability		0.002						
T_0	initial temperature			1000000					1000
SA_{max}	number of iterations			1000					
α	cooling rate			0.99					0.90
β_{min}	minimum rate of shaking			0.005	0.005	0.005			0.10
β_{max}	maximum rate of shaking			0.05	0.10				0.30
k_{max}	number of neighborhoods					10			
h_s	start grid dense						0.125		
h_e	end grid dense						0.00012		
c_1	cognitive coefficient							2.05	
c_2	social coefficient							2.05	
w	inertia weight							0.73	

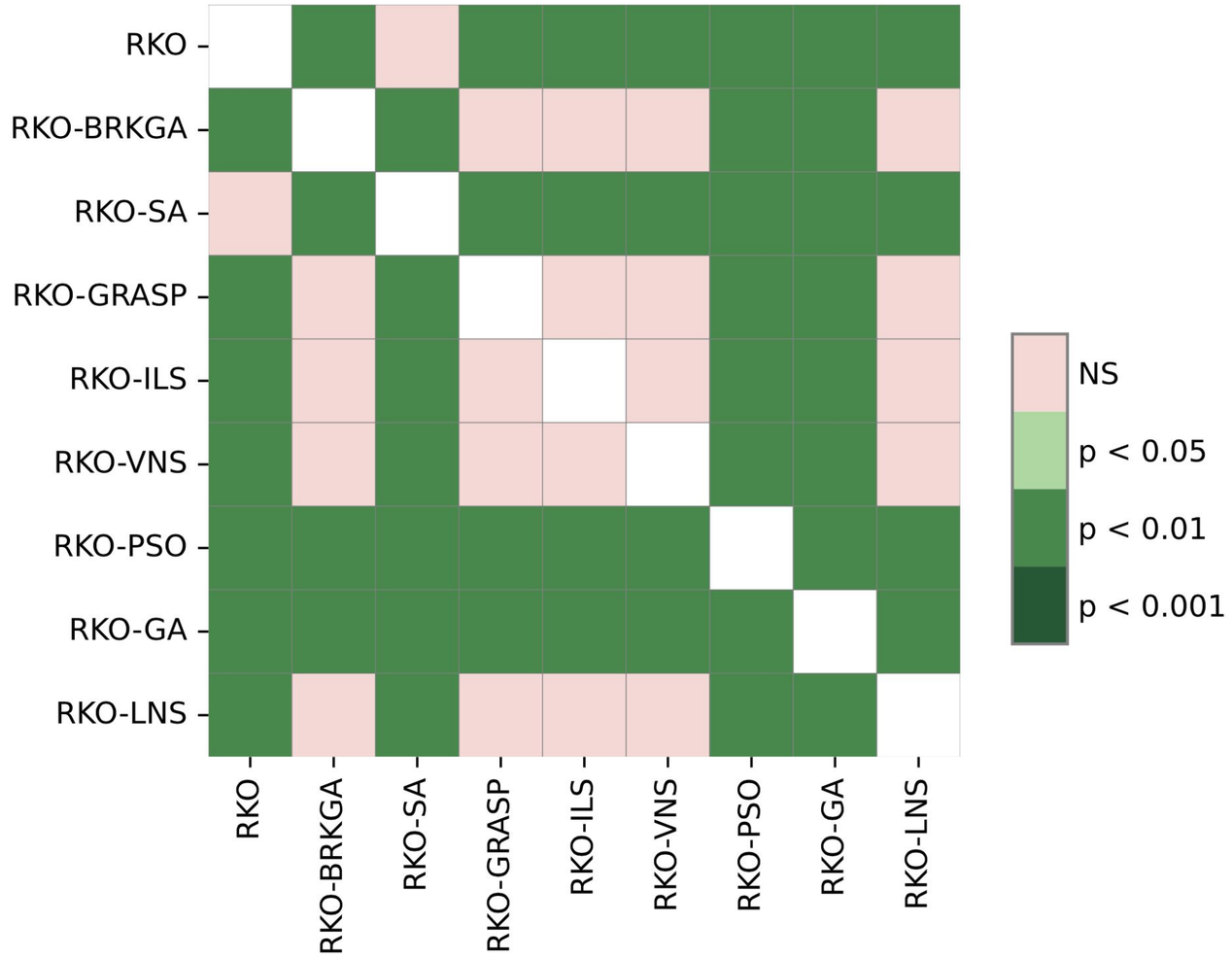
Node Capacitated Graph Partitioning Problem

Summary of the NCGPP results for the benchmark set with 83 instances.

Method	*Best*	RPD_{best}	RPD_{aver}	best found at (s)	*#BKS*
Gurobi	232534.93	24.39	-	487.83	49
GRASP[1]	157464.01	4.46	-	-	47
GevPR[2]	139098.19	0.64	-	-	47
BRKGA[2]	142737.82	1.69	-	-	37
RKO	136605.66	0.01	0.08	70.71	**76**
RKO-BRKGA	137395.28	0.20	11.11	75.91	56
RKO-SA	136680.94	0.02	0.12	73.04	72
RKO-GRASP	137690.41	0.21	0.55	91.52	59
RKO-ILS	137994.82	0.29	0.56	80.59	55
RKO-VNS	137701.45	0.21	0.56	78.79	58
RKO-PSO	140043.08	0.91	1.60	27.22	42
RKO-GA	137656.60	0.27	2.18	66.81	50
RKO-LNS	137827.69	0.26	0.50	82.79	58

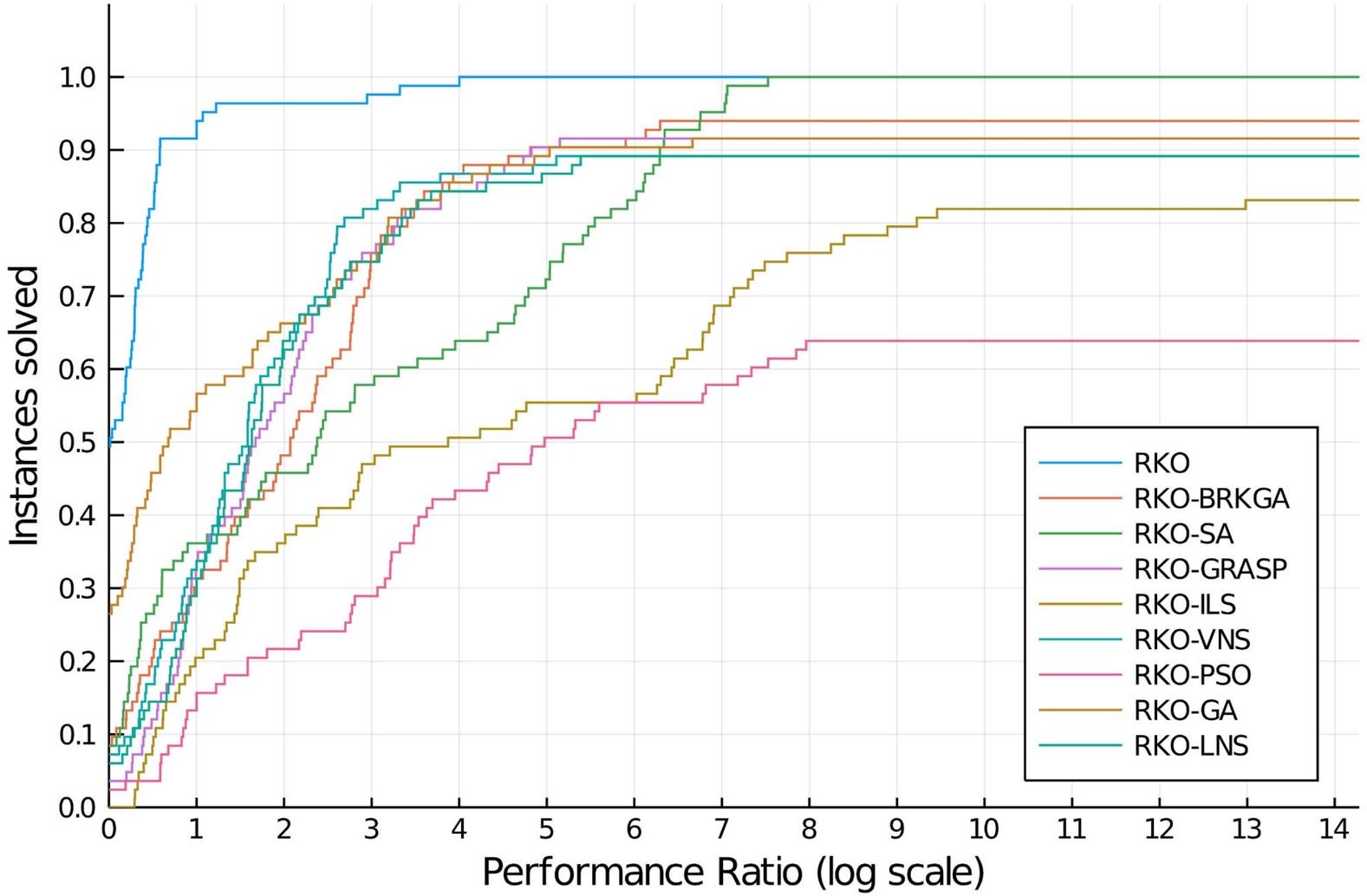
Mateus et al. (2011)[1] Morán-Mirabal et al. (2013)[2]

Node Capacitated Graph Partitioning Problem



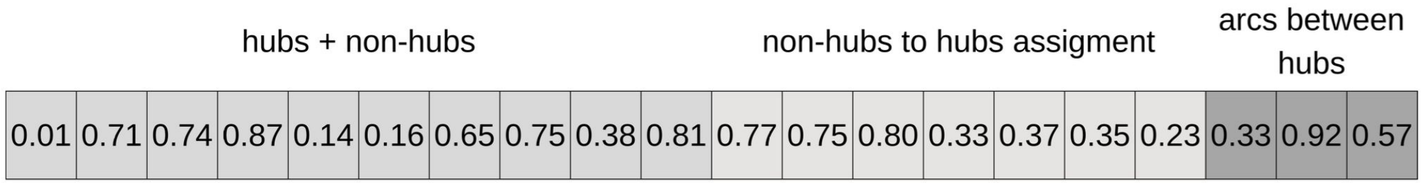
Heat map of the results (p-values) of the Nemenyi test for the RKO methods to solve the NCGPP.

Node Capacitated Graph Partitioning Problem

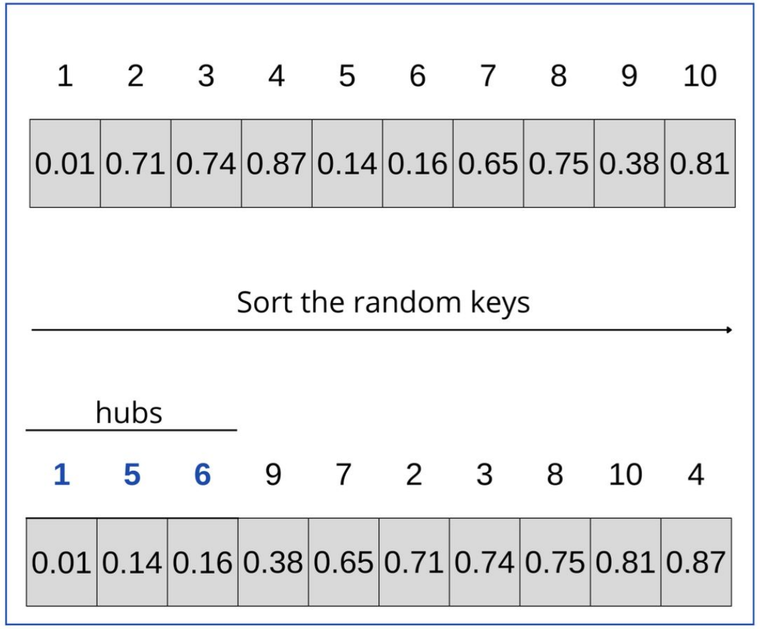


Performance profile of runtime for RKO methods to solve the NCGPP.

Tree Hub Location Problem



Encoding

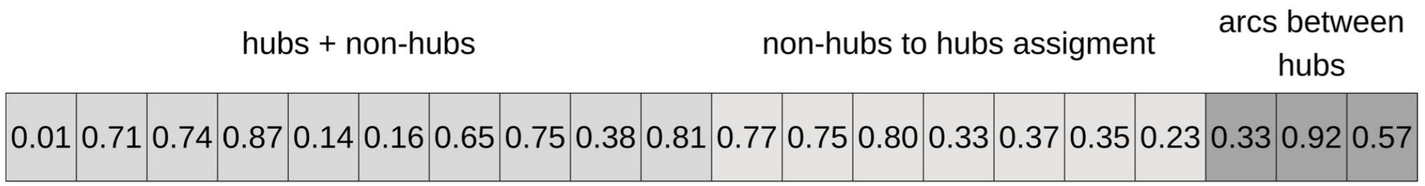


$|N| = 10, p = 3$

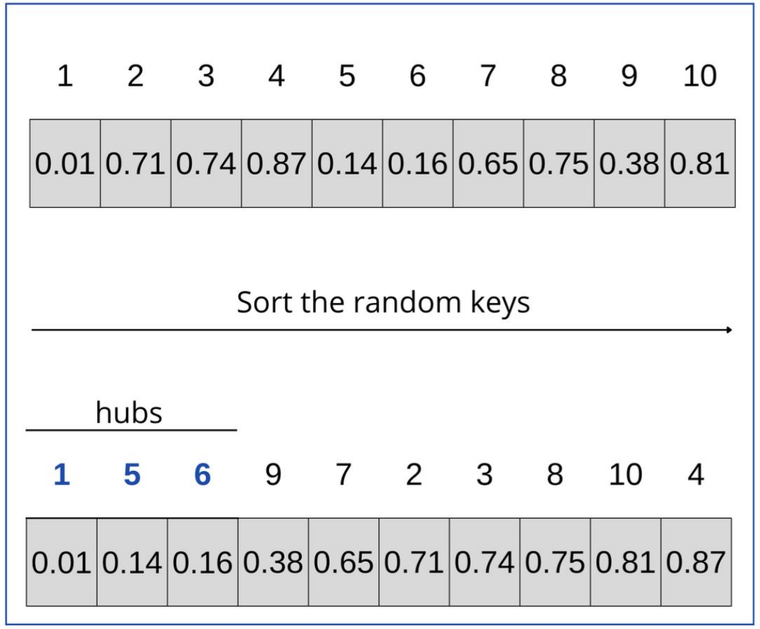
Decoder

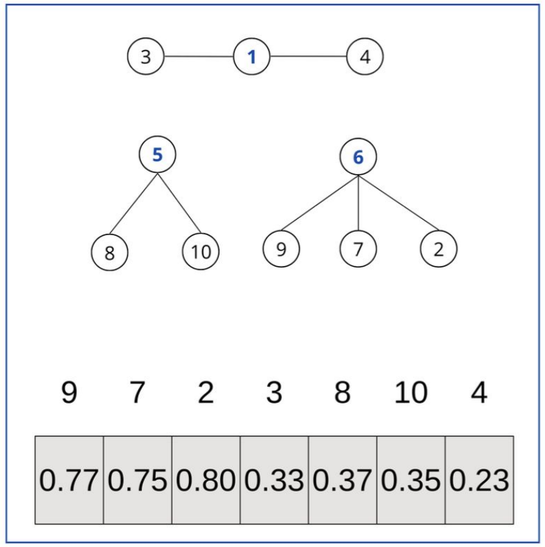
(Pessoa et al., 2017)

Tree Hub Location Problem



Encoding



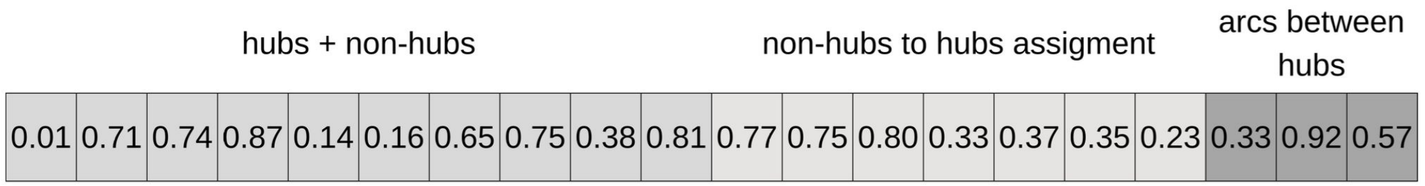


$|N| = 10, p = 3$

Decoder

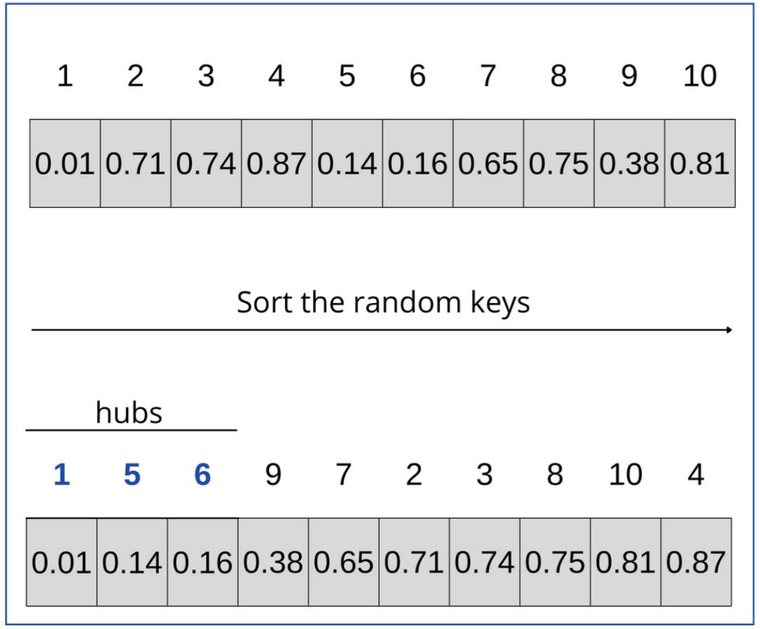
(Pessoa et al., 2017)

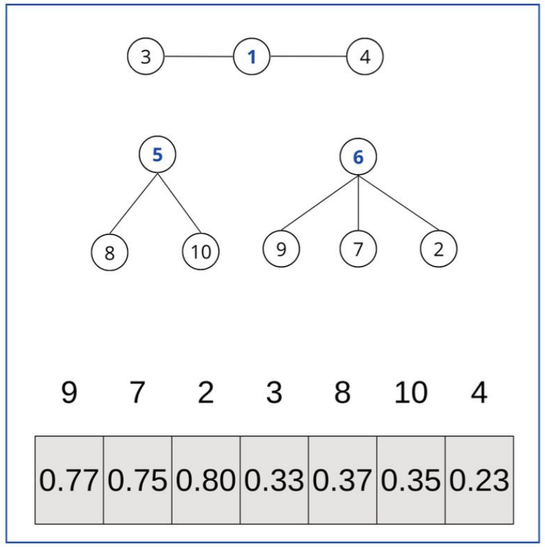
Tree Hub Location Problem



Encoding

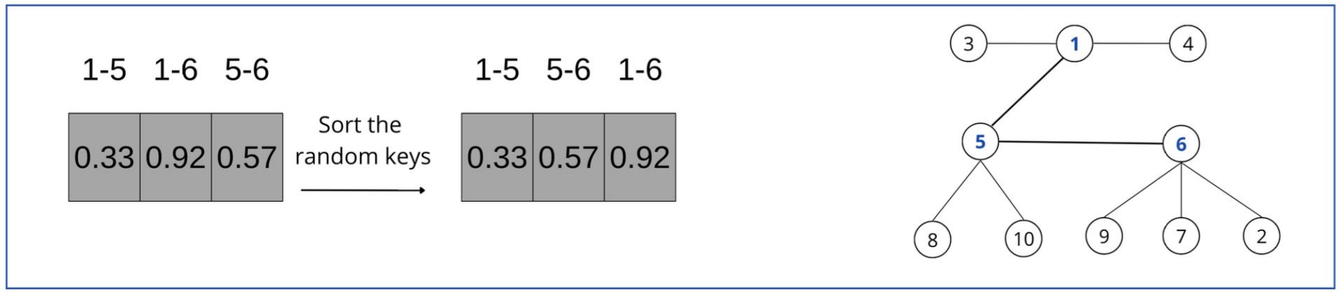
$|N| = 10, p = 3$





Decoder

(Pessoa et al., 2017)



Tree Hub Location Problem

Parameters of the RKO metaheuristics to solve the THLP.

Parameters	Definition	BRKGA	GA	SA	ILS	VNS	GRASP	PSO	LNS
p	population size	1597	600					200	
p_e	elite set	0.15							
p_m	mutant set	0.20							
ρ	inherit probability	0.70							
p_c	crossover probability		0.99						
μ	mutation probability		0.005						
T_0	initial temperature			1000000					1000000
SA_{max}	number of iterations			1500					
α	cooling rate			0.99					0.97
β_{min}	minimum rate of shaking			0.01	0.05	0.005			0.10
β_{max}	maximum rate of shaking			0.05	0.20				0.30
k_{max}	number of neigborhoods					10			
h_s	start grid dense						0.125		
h_e	end grid dense						0.00012		
c_1	cognitive coefficient							2.05	
c_2	social coefficient							2.05	
w	inertia weight							0.73	

Tree Hub Location Problem

Summary of the THLP results for the CAP and AP small instances.

Method	*Best*	RPD_{best}	RPD_{aver}	best found at (s)	$\#BKS$
Exact method[1]	26620.58	0.04	-	1436.51	59
BRKGA[2]	26685.72	0.22	-	53.93	45
RKO	26608.76	0.00	0.04	1.04	**63**
RKO-BRKGA	26826.06	0.66	1.59	4.19	37
RKO-SA	26609.03	0.00	0.05	3.54	62
RKO-GRASP	26608.76	0.00	0.04	1.69	**63**
RKO-ILS	26608.76	0.00	0.07	1.29	**63**
RKO-VNS	26608.76	0.00	0.00	1.08	**63**
RKO-PSO	26608.76	0.00	0.00	1.69	**63**
RKO-GA	26612.17	0.02	0.09	3.64	58
RKO-LNS	26608.76	0.00	0.00	1.30	**63**

Contreras et al. (2009, 2010)[1] Pessoa et al. (2017)[2]

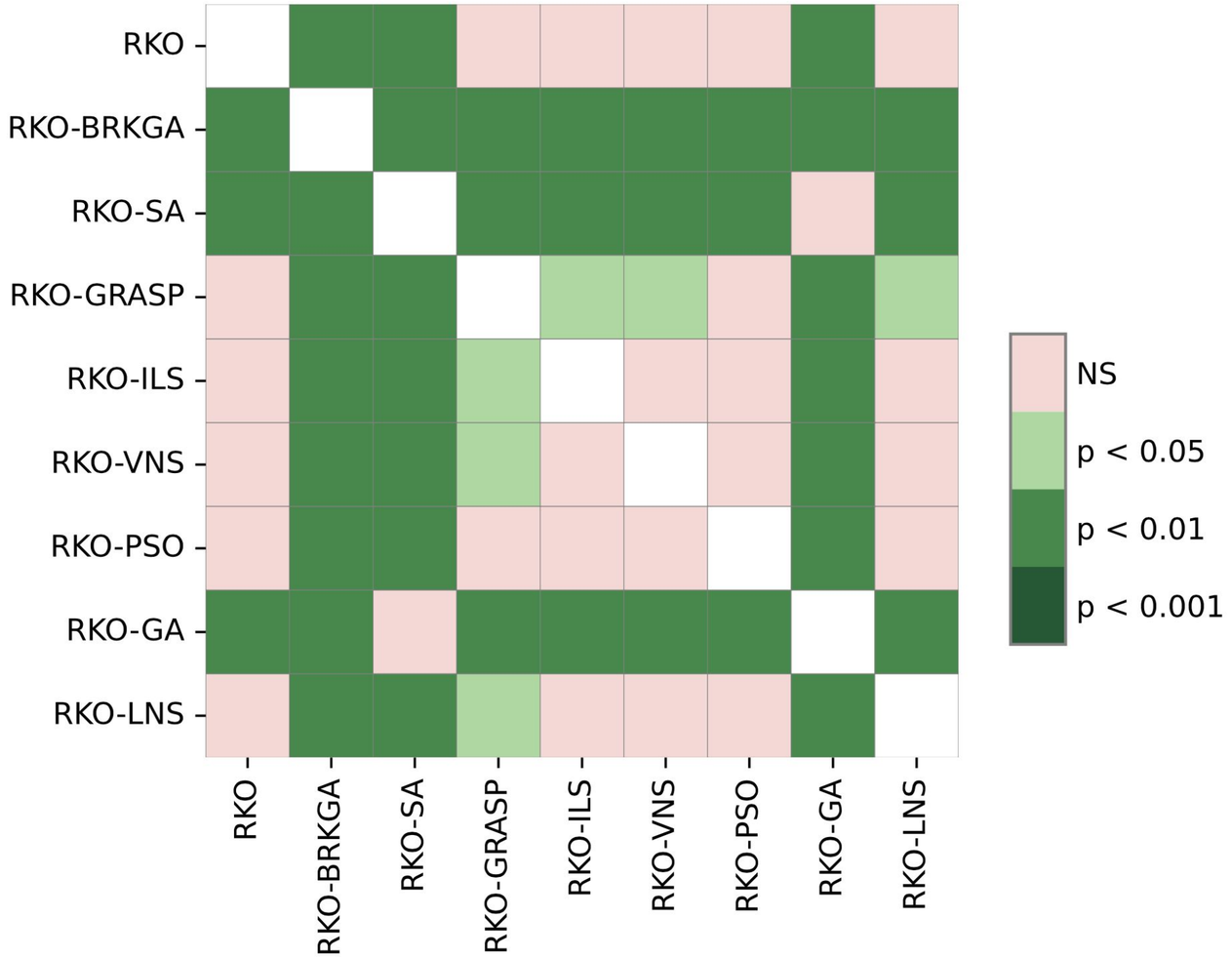
Tree Hub Location Problem

Summary of the THLP results for the AP medium and large instances.

Method	*Best*	RPD_{best}	RPD_{aver}	best found at (s)	$\#BKS$
Primal heuristic[1]	66203.51	0.89	-	1616.26	29
RKO	65639.39	0.005	0.12	16.75	**60**
RKO-BRKGA	67281.20	2.71	5.10	9.62	1
RKO-SA	65940.80	0.46	1.02	30.04	35
RKO-GRASP	65651.48	0.03	0.31	28.64	56
RKO-ILS	65642.42	0.009	0.57	21.53	58
RKO-VNS	65642.51	0.009	0.12	27.26	59
RKO-PSO	65683.33	0.07	0.20	24.90	48
RKO-GA	65777.66	0.22	0.71	30.06	34
RKO-LNS	65642.73	0.01	0.10	23.66	59

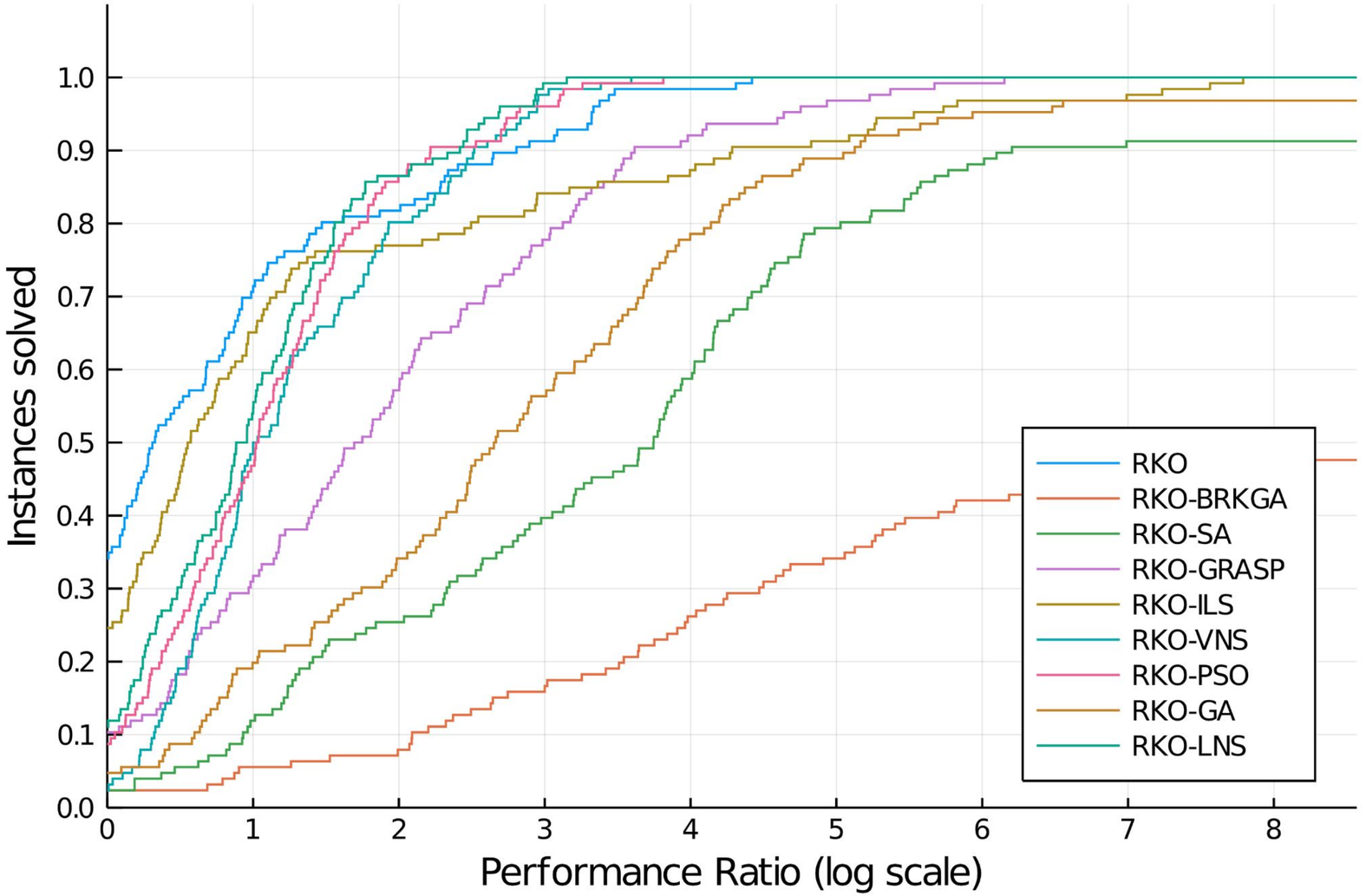
Contreras et al. (2009)[1]

Tree Hub Location Problem



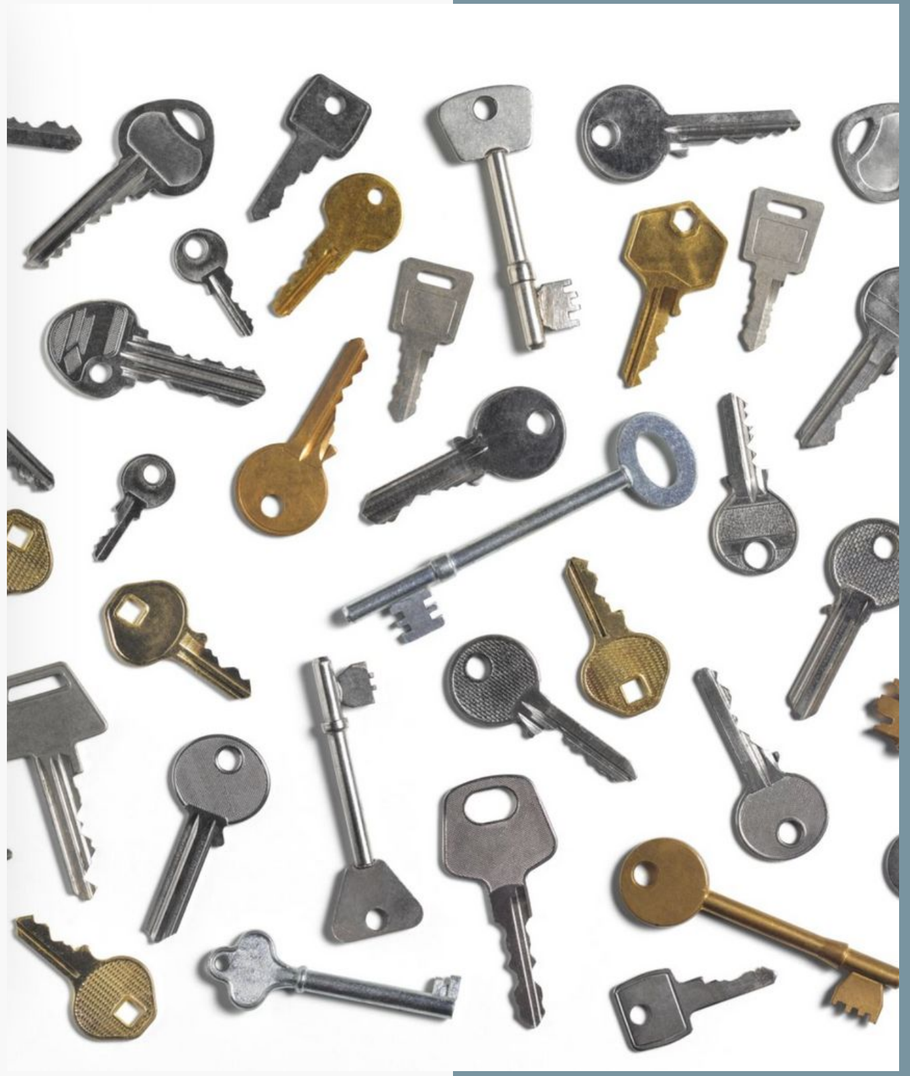
Heat map of the results (p-values) of the Nemenyi test for the RKO methods to solve the THLP.

Tree Hub Location Problem



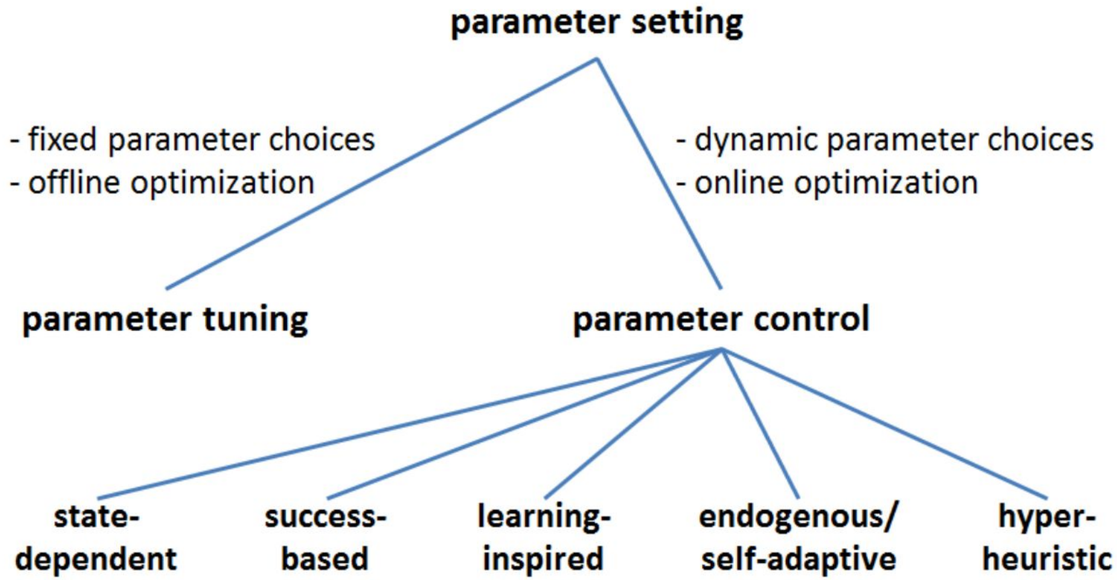
Performance profile of runtime for RKO methods to solve the THLP.

Parameter Setting



Parameter Setting

One **disadvantage** of metaheuristics in general is the large number of parameters to be set; often, automatic parameter tuning tools are necessary to effectively and efficiently find a good set of values.



Reinforcement Learning

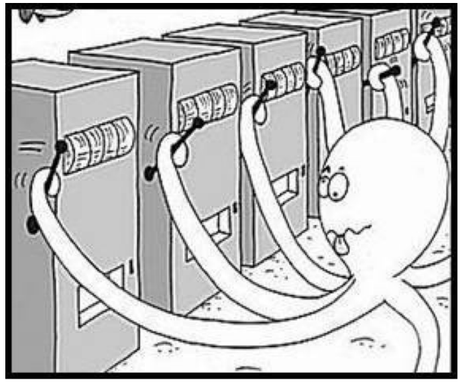
- The agent “**experiments**” with a system, and the system responds to this experimentation with a **reward** or **punishments**;
- The agent optimizes its behaviour, its **goal** being to **maximize the rewards** and to **minimizes the punishments**;
- Simultaneously: **learn by trial-and-error, learn a model of the environment, and use the model for planning.**

Reinforcement Learning

- **N-Armed Bandit**

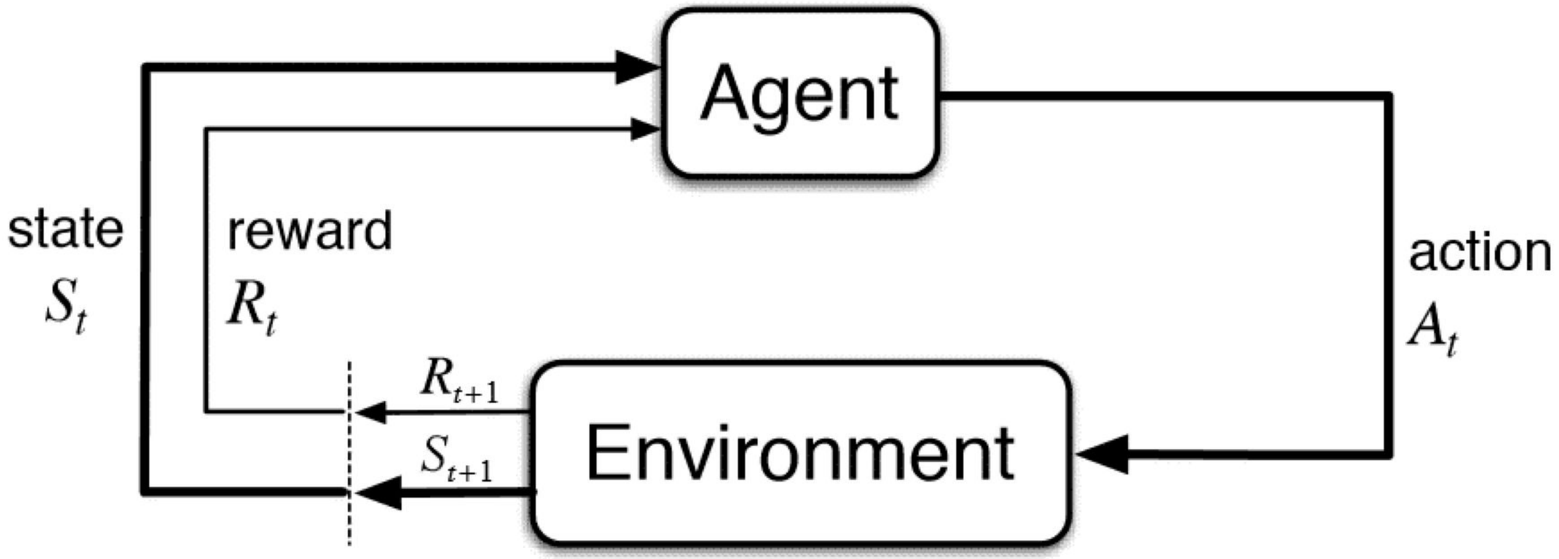
  What of the configurations offers

  the highest average return?

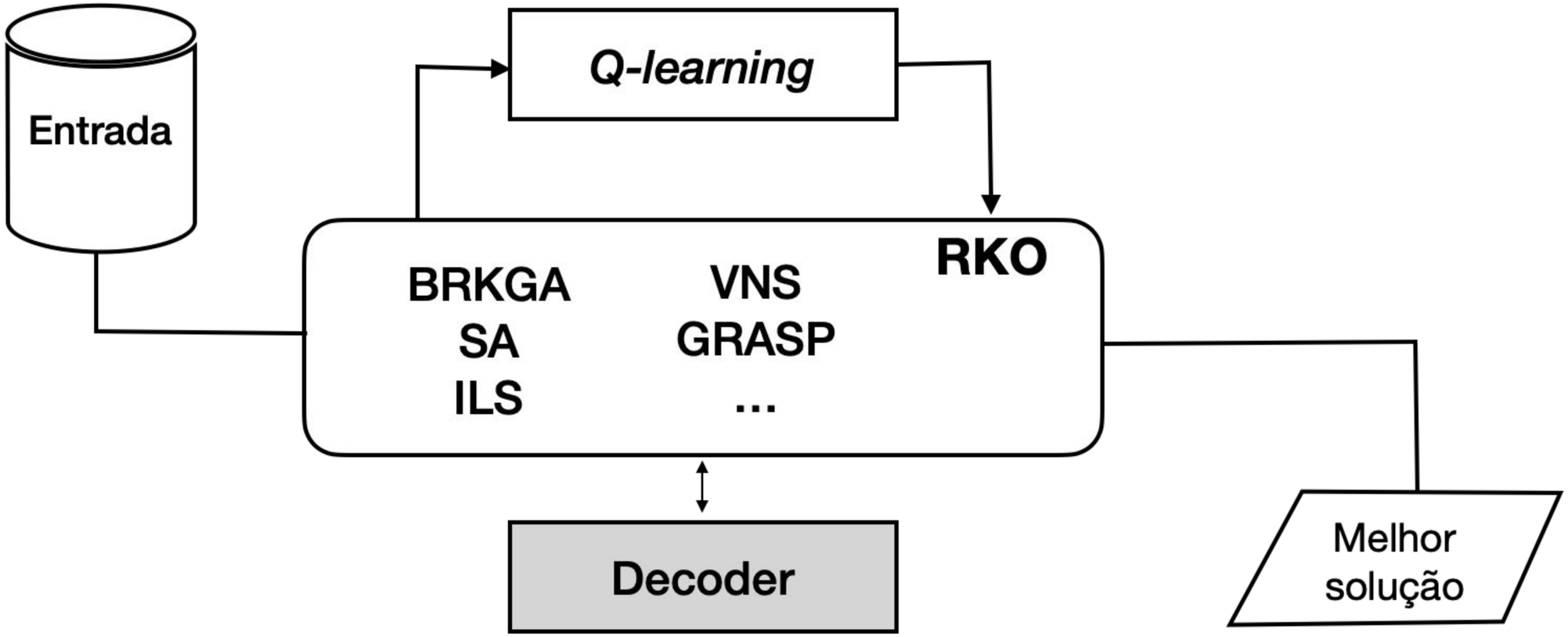


- **Strategy**
  - Make an initial choice
  - Occasionally experiments with other configurations
  - If better, replace the “previously best”
  - Exploitation (“best” configuration) x Exploration (“new” configuration)

Q-Learning (Watkins, 1989)



RKO + Q-Learning

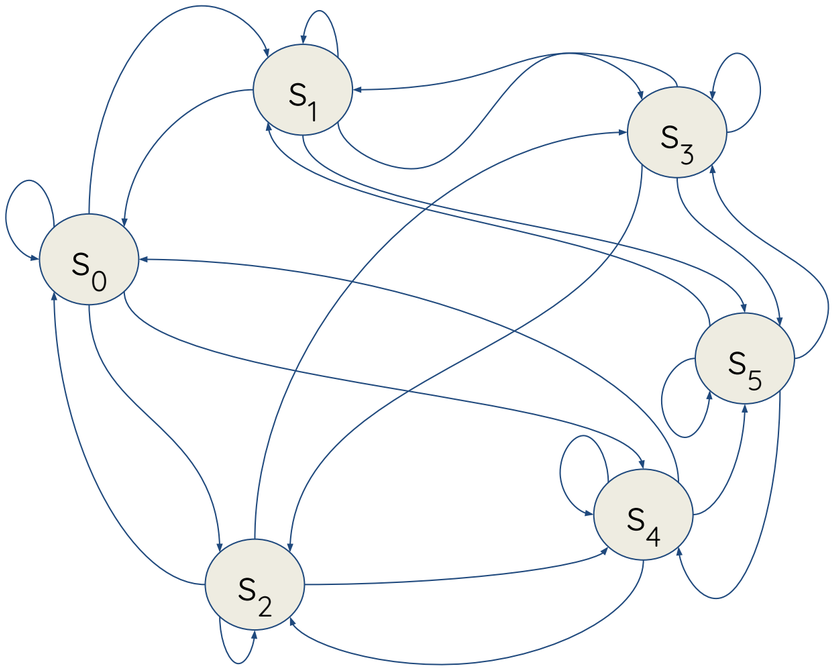


RKO + Q-Learning

- **Markovian Decision Process (Bellman, 1957)**
  - S = [s_1, s_2, ..., s_m] => finite set of states (parameters choices)
  - A = [a_1, a_2, ..., a_k] => finite set of actions that represents move to other states
  - Q(s_i , a_j) represents the quality of the choice [s_i , a_j]
  - In each step *t* :
    - Choose an action $\boldsymbol{a_t}$ from $\boldsymbol{s_t}$ with **ε-Greed policy**
    - Compute this action and receive a positive or negative **reward** $\boldsymbol{r_{t+1}}$
    - Update Q(s_t, a_t)
    - Move to new state $\boldsymbol{s_{t+1}}$
  - The agent aims to **maximize its total reward**

$$\boldsymbol{Q(s_t,a_t) = Q(s_t,a_t) + lf \times [\, r_{t+1} + df \times max\, Q(s_{t+1}, a) - Q(s_t,a_t)]}$$

Markovian Decision Process - Example of ILS



States

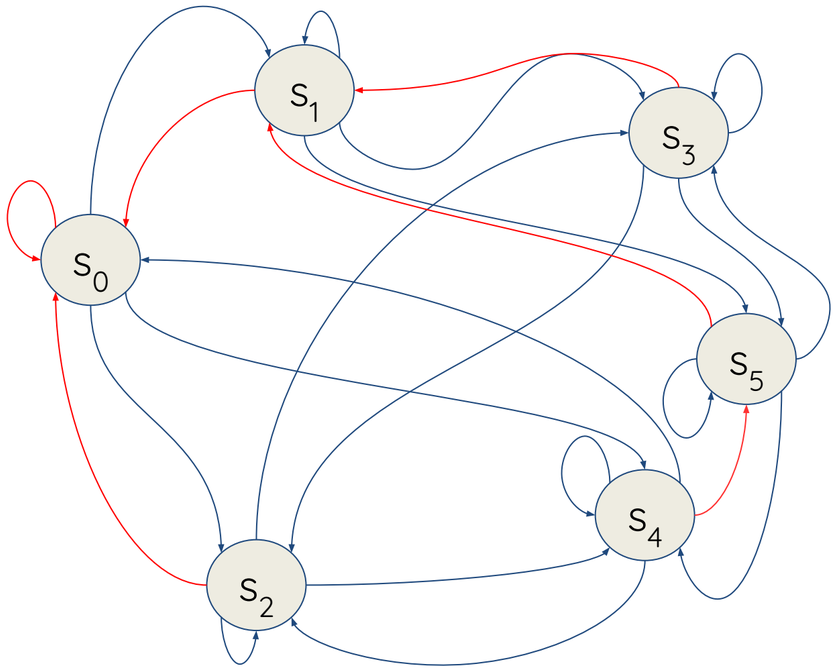
s_0	0.05	0.20
s_1	0.05	0.25
s_2	0.10	0.20
s_3	0.10	0.25
s_4	0.15	0.20
s_5	0.15	0.25

$\beta_{min} = \{0.05, 0.10, 0.15\}$
$\beta_{max} = \{0.20, 0.25\}$

Value Function $Q(s_i, a_j)$

s_0:	0 (0.00)	1 (0.00)	2 (0.00)	4 (0.00)
s_1:	0 (0.00)	1 (0.00)	3 (0.00)	5 (0.00)
s_2:	0 (0.00)	2 (0.00)	3 (0.00)	4 (0.00)
s_3:	1 (0.00)	2 (0.00)	3 (0.00)	5 (0.00)
s_4:	0 (0.00)	2 (0.00)	4 (0.00)	5 (0.00)
s_5:	1 (0.00)	3 (0.00)	4 (0.00)	5 (0.00)

Markovian Decision Process - Example of ILS



States

s_0	0.05	0.20
s_1	0.05	0.25
s_2	0.10	0.20
s_3	0.10	0.25
s_4	0.15	0.20
s_5	0.15	0.25

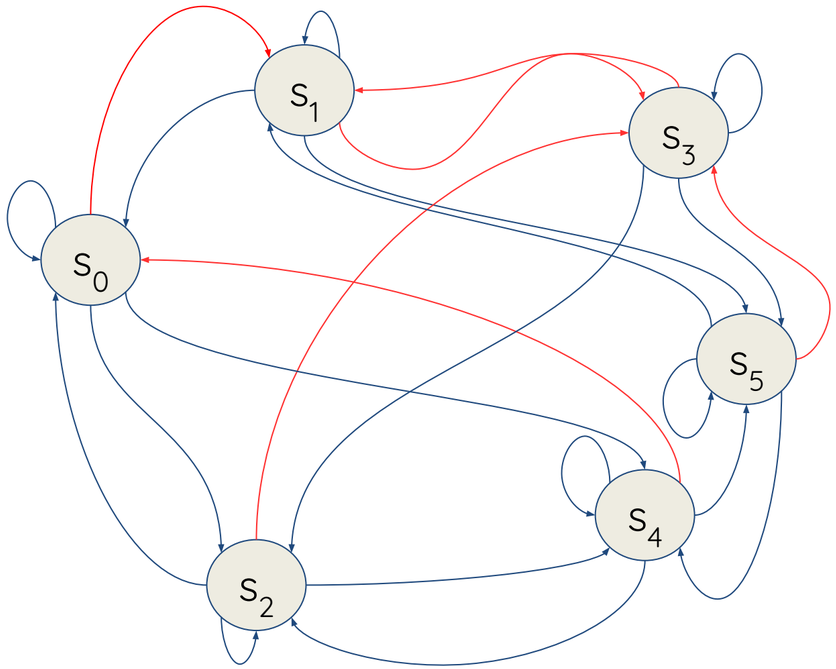
β_{min} = {0.05, 0.10, 0.15}
β_{max} = {0.20, 0.25}

Value Function $Q(s_i, a_j)$

s_0:	**0 (0.63)**	1 (-0.01)	2 (-0.02)	4 (0.03)
s_1:	**0 (0.21)**	1 (-0.03)	3 (-0.05)	5 (-0.02)
s_2:	**0 (0.51)**	2 (-0.01)	3 (0.02)	4 (0.07)
s_3:	**1 (0.67)**	2 (0.24)	3 (0.20)	5 (0.11)
s_4:	0 (0.03)	2 (-0.07)	4 (0.05)	**5 (0.38)**
s_5:	**1 (1.44)**	3 (-0.03)	4 (0.06)	5 (0.13)

Policy found for instance d198.tsp

Markovian Decision Process - Example of ILS



States

s_0	0.05	0.20
s_1	0.05	0.25
s_2	0.10	0.20
s_3	0.10	0.25
s_4	0.15	0.20
s_5	0.15	0.25

β_{min} = {0.05, 0.10, 0.15}
β_{max} = {0.20, 0.25}

Value Function $Q(s_i, a_j)$

s_0:	0 (0.03)	**1 (0.47)**	2 (0.01)	4 (0.10)
s_1:	0 (0.10)	1 (0.04)	**3 (0.44)**	5 (0.02)
s_2:	0 (-0.02)	2 (0.03)	**3 (2.02)**	4 (0.51)
s_3:	**1 (0.69)**	2 (0.05)	3 (0.09)	5 (0.40)
s_4:	**0 (0.57)**	2 (0.06)	4 (-0.02)	5 (-0.03)
s_5:	1 (0.87)	**3 (0.94)**	4 (0.54)	5 (0.70)

Policy found for instance lin318.tsp

Choose an action

- Choose an action (value) from Q-Table for each state (parameter) using the **ε-greedy policy**
- We must establish a compromise between visiting more often good actions (**intensification**) and explore eventually actions that can lead to the discovery of new policies even better than those already existing (**diversification**)

$$\pi(s) = \begin{cases} a^*, & \text{with probability } 1 - \varepsilon, \\ \textit{any other}, & \text{with probability } \varepsilon \end{cases}$$

Q-Learning parameters

- Restart ε once T computational time are performed (T = 0.1 x runtime)
  - Cosine annealing decay with warm restart:

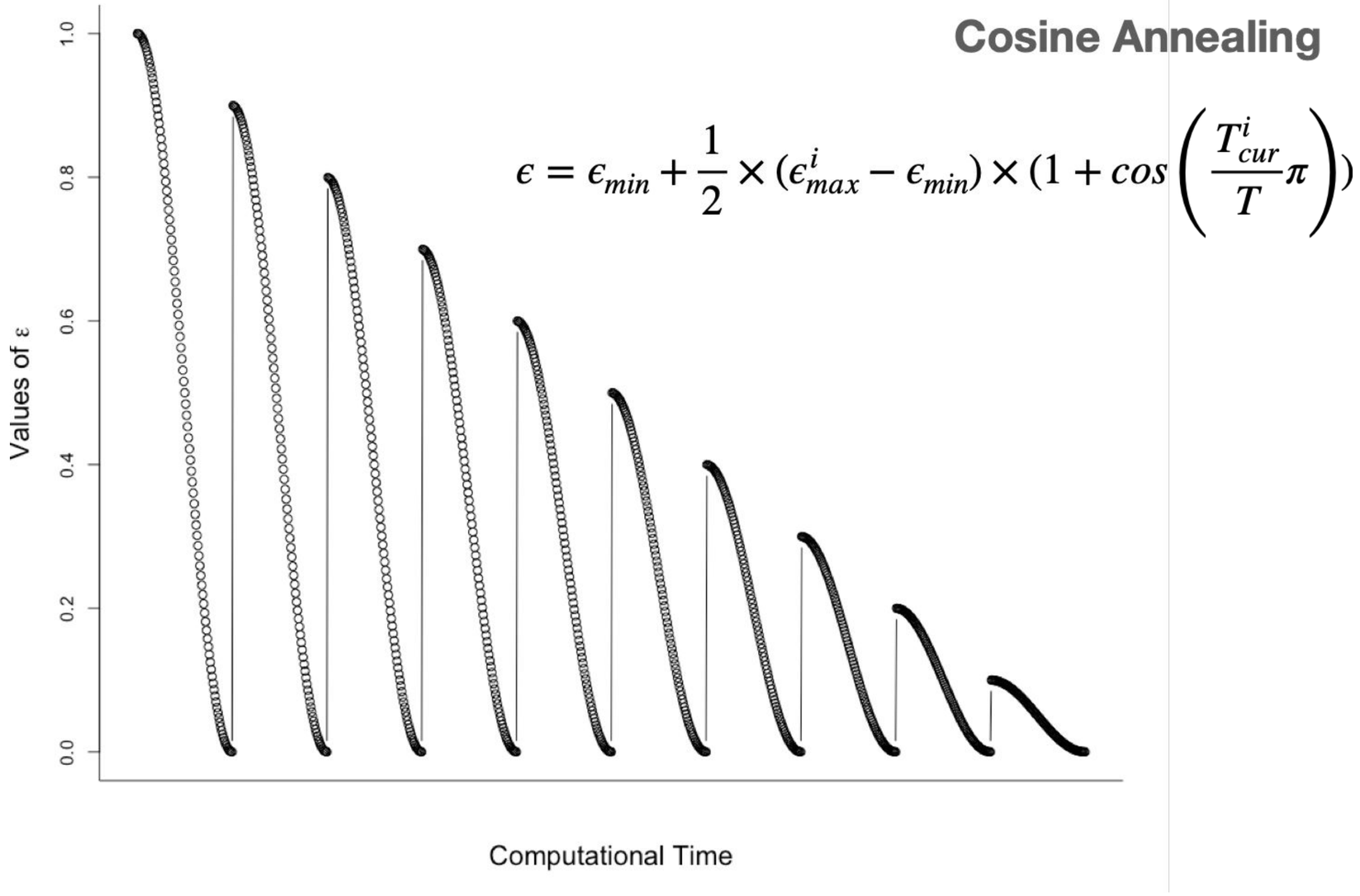
$$\epsilon = \epsilon_{min} + \frac{1}{2} \times (\epsilon^i_{max} - \epsilon_{min}) \times (1 + cos\left(\frac{T^i_{cur}}{T}\pi\right))$$

- Learning factor
  - Initially, a higher priority is given to the newly gained information (exploration mode). Then, we decrement lf and have a higher priority for the existing information in Q-Table (exploitation mode)

$$lf = 1 - (0.9 \times \frac{\sum_i T^i_{cur}}{run_{time}})$$

- Discount factor
  - We look for a higher, long-term reward: $df = 0.8$

Decay with a cosine annealing and warm restart



Update Q-Table

- Reward:

$$r_{t+1} = \begin{cases} 1, & \textbf{if } f_b^{t+1} < f_b^t \\ \frac{f_b^t - f_b^{t+1}}{f_b^{t+1}}, & otherwise \end{cases}$$

- The function Q(s, a) is the value associated with the state-action pair (s, a) and represents how good is the choice of this action on optimising the expected total return:

$$Q(s_t, a_t) = Q(s_t, a_t) + lf \times [\, r_{t+1} + df \times max\, Q(s_{t+1}, a) - Q(s_t, a_t)]$$

- Q(s, a) is incremented when the action leads to a state, in which there exists an action such that the best possible Q-value and the reward R is greater than current value of Q(s, a), i.e., the old value of Q(s, a) was too pessimistic;

Example of RKO run with Q-Learning

Instance: /TSP/d198.tsp

Thread 0 executing MH_0 BRKGA.

Thread 1 executing MH_1 SA.

Thread 2 executing MH_2 GRASP.

Thread 3 executing MH_3 ILS.

Thread 4 executing MH_4 VNS.

Thread 5 executing MH_5 PSO.

Thread 6 executing MH_6 GA.

Thread 7 executing MH_7 LNS.

Thread 8 executing MH_8 BRKGA_CS.

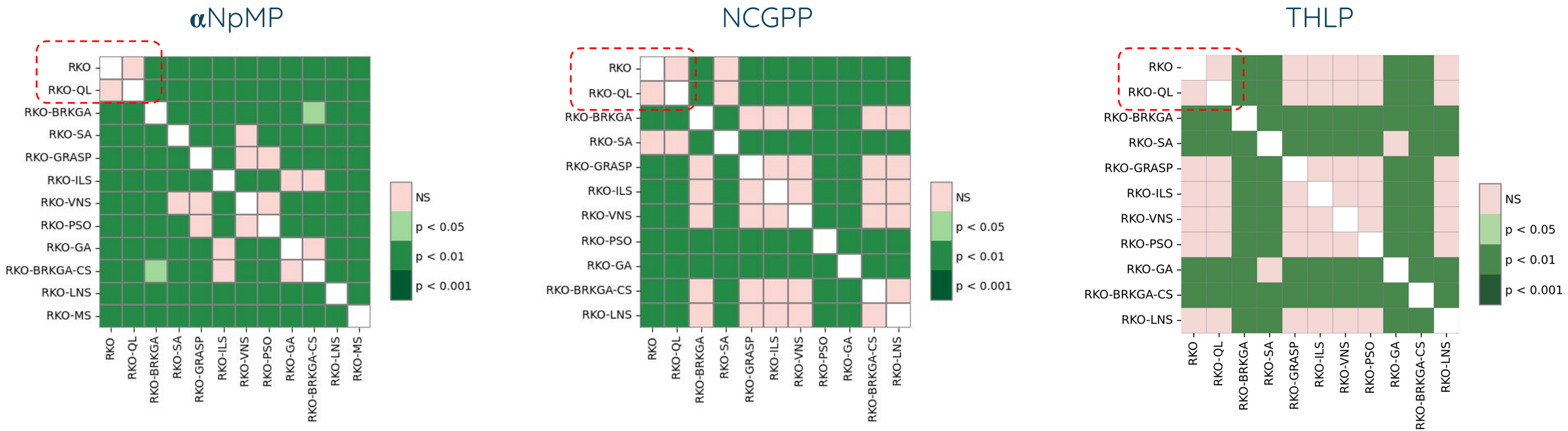
Run 1

Best solution: 16031.00 (MH: 6 - Thread: 6)
Best solution: 15904.00 (MH: 7 - Thread: 7)
Best solution: 15860.00 (MH: 6 - Thread: 6)
Best solution: 15832.00 (MH: 8 - Thread: 8)
Best solution: 15824.00 (MH: 2 - Thread: 2)
Best solution: 15804.00 (MH: 6 - Thread: 6)
Best solution: 15802.00 (MH: 1 - Thread: 1)
Best solution: 15799.00 (MH: 1 - Thread: 1)
Best solution: 15797.00 (MH: 1 - Thread: 1)
Best solution: 15795.00 (MH: 1 - Thread: 1)
Best solution: 15793.00 (MH: 1 - Thread: 1)
Best solution: 15785.00 (MH: 0 - Thread: 0)
Best solution: 15782.00 (MH: 7 - Thread: 7)
Best solution: 15780.00 (MH: 6 - Thread: 6)

Run 2

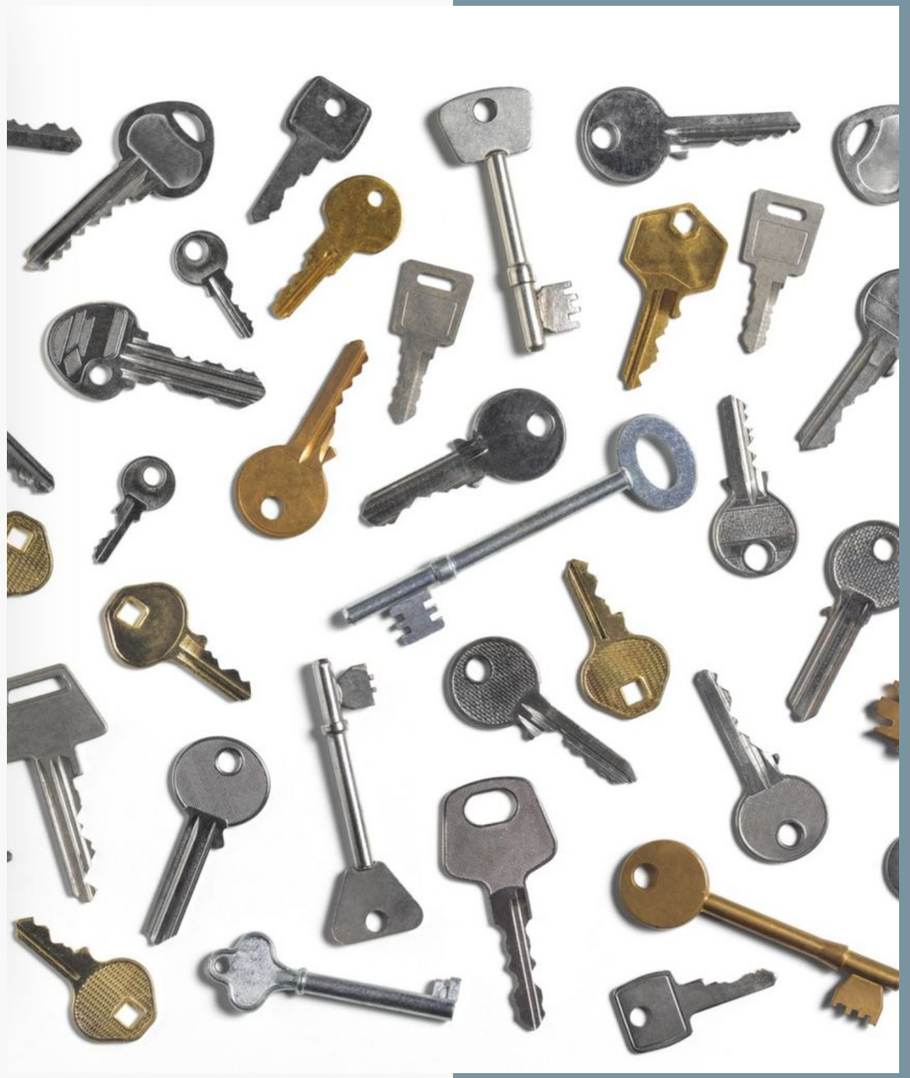
Best solution: 16224.00 (MH: 1 - Thread: 1)
Best solution: 16117.00 (MH: 6 - Thread: 6)
Best solution: 16083.00 (MH: 1 - Thread: 1)
Best solution: 16072.00 (MH: 1 - Thread: 1)
Best solution: 15913.00 (MH: 6 - Thread: 6)
Best solution: 15848.00 (MH: 0 - Thread: 0)
Best solution: 15816.00 (MH: 5 - Thread: 5)
Best solution: 15809.00 (MH: 4 - Thread: 4)
Best solution: 15807.00 (MH: 8 - Thread: 8)
Best solution: 15802.00 (MH: 2 - Thread: 2)
Best solution: 15800.00 (MH: 0 - Thread: 0)
Best solution: 15795.00 (MH: 6 - Thread: 6)
Best solution: 15789.00 (MH: 2 - Thread: 2)
Best solution: 15784.00 (MH: 2 - Thread: 2)
Best solution: 15780.00 (MH: 0 - Thread: 0)

Results of RKO run with Q-Learning



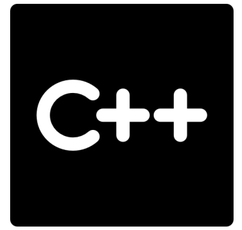
Heat map of the results (p-values) of the Nemenyi test for the RKO methods to solve the each problem.

RKO API



RKO API

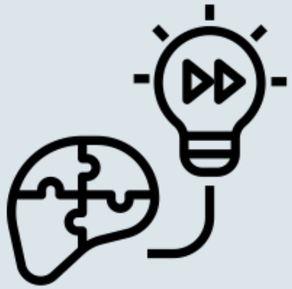
https://github.com/antoniochaves19/RKO



RKO API

Main

- Main
- Data
- Global Variables
- Global Functions
- Output



MH

- Methods
- MH-s ...
- Q-Learning

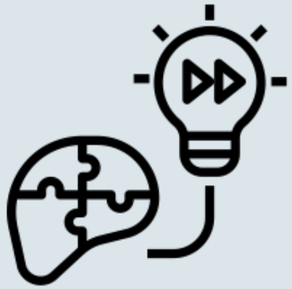
Problem

- ReadData
- Decoder

RKO API

Main

- Main
- Data
- Global Variables
- Global Functions
- Output



MH

- Methods
- MH-s ...
- Q-Learning

Problem

- ReadData
- Decoder

Problem independent

Problem dependent

RKO API

Running the algorithm

a) Enter the Program directory: cd Program

b) Run the make command: **make rebuild**

c) Run the RKO:

GNU Make

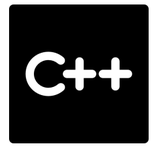
File **testScenario.csv** is the input data problem, and each line consists of:

Instance Name
Maximum runtime (in seconds)
Maximum number of runs

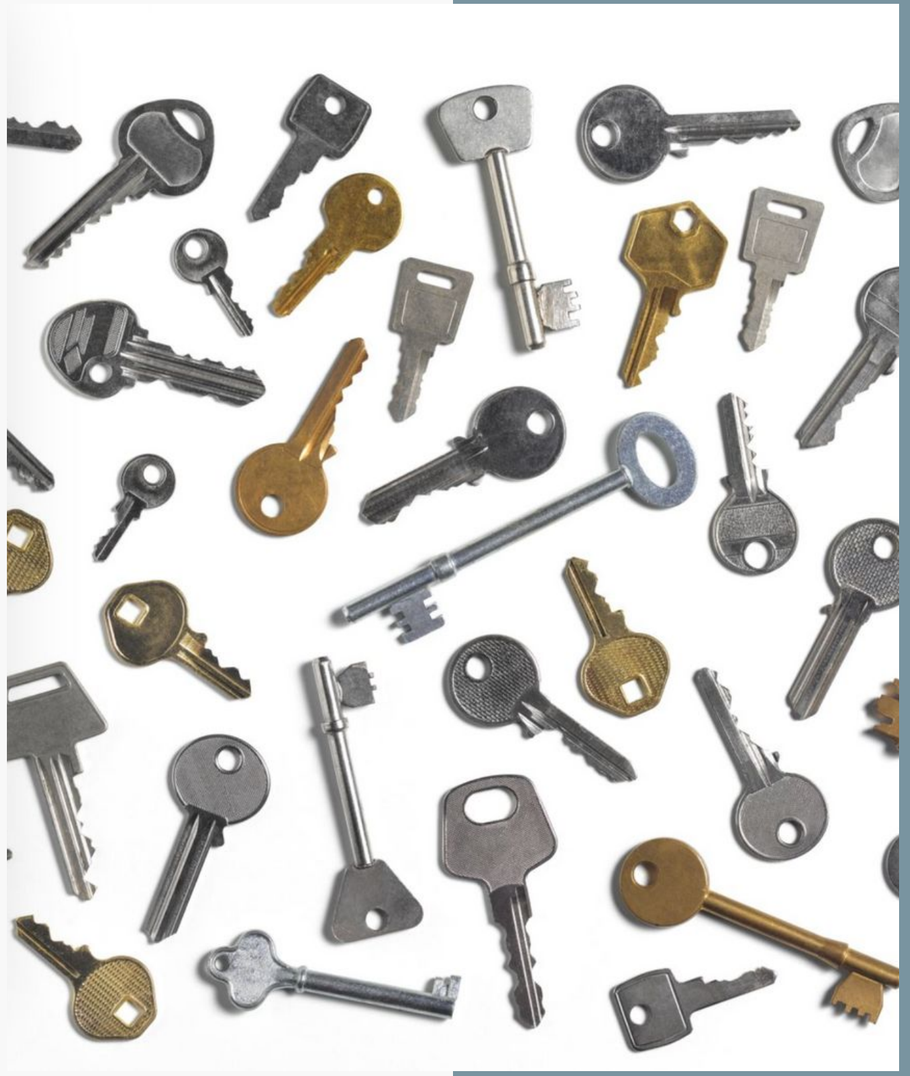
./runTest ../Instances/testScenario.csv M T D

where M is the number of metaheuristics used to solve the problem (9 to RKO), T is the tuning method (0 is offline and 1 is online), and D is the debug mode (1 to print information in the screen and 0 to save information in file).

Or compile via terminal: g++ -o runTest Main/main.cpp -O3 -fopenmp



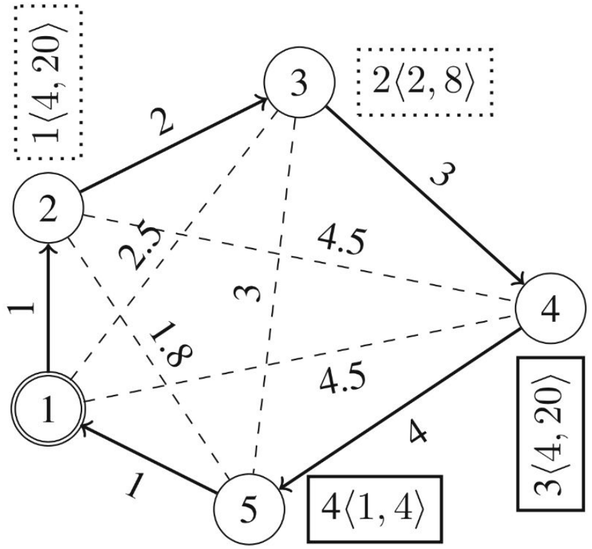
Travelling Thief Problem Challenge



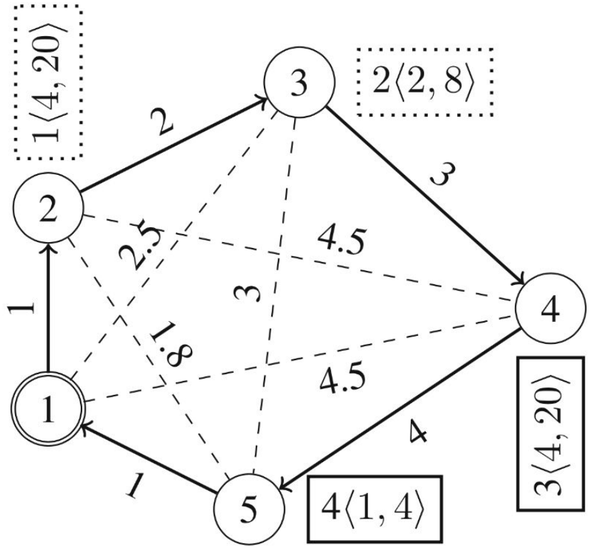
Travelling Thief Problem - Challenge

Travelling Thief Problem - Challenge

Encoding	2	3	4	5	1	2	3	4	P
	0.3	0.5	0.9	0.6	0.4	0.8	0.1	0.6	0.5



Travelling Thief Problem - Challenge



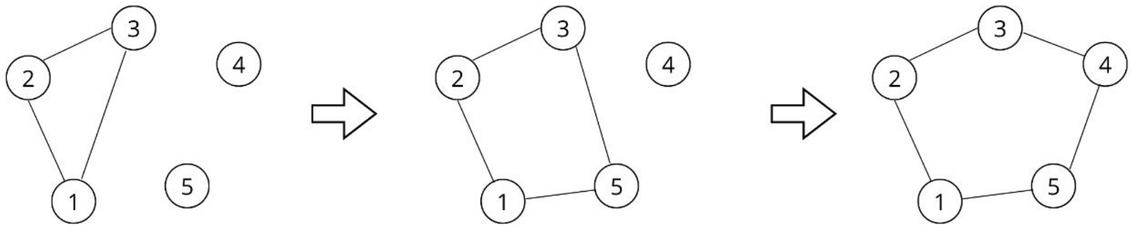
Encoding

2	3	4	5	1	2	3	4	P
0.3	0.5	0.9	0.6	0.4	0.8	0.1	0.6	0.5

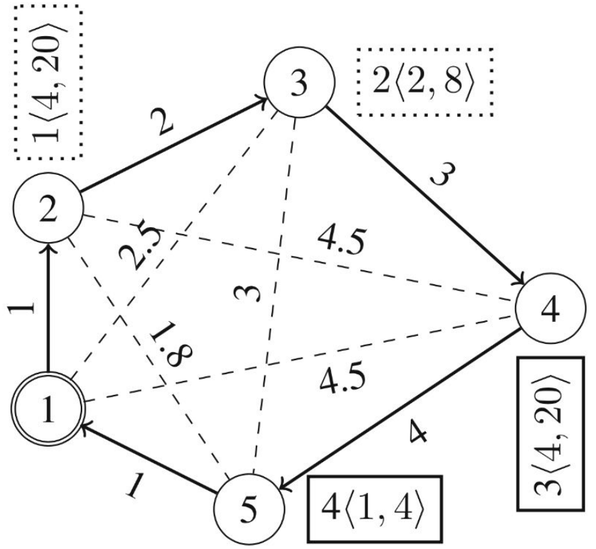
Decoder Sort the random keys →

2	3	5	4
0.3	0.5	0.6	0.9

Cheapest Insertion heuristic



Travelling Thief Problem - Challenge



Encoding

2	3	4	5	1	2	3	4	P
0.3	0.5	0.9	0.6	0.4	0.8	0.1	0.6	0.5

Decoder Sort the random keys →

2	3	5	4	1	2	3	4	P
0.3	0.5	0.6	0.9	0.4	0.2	0.9	0.6	0.5

Collection plan heuristic

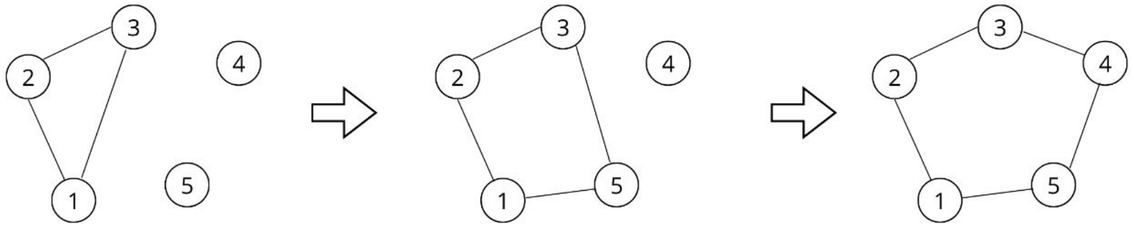
PW = profit / weight

0	1	2	3
8/2	4/1	20/4	20/4

0.5 x 4 = 2
PW[2] = 5

{0, 0, 1, 1}

Cheapest Insertion heuristic



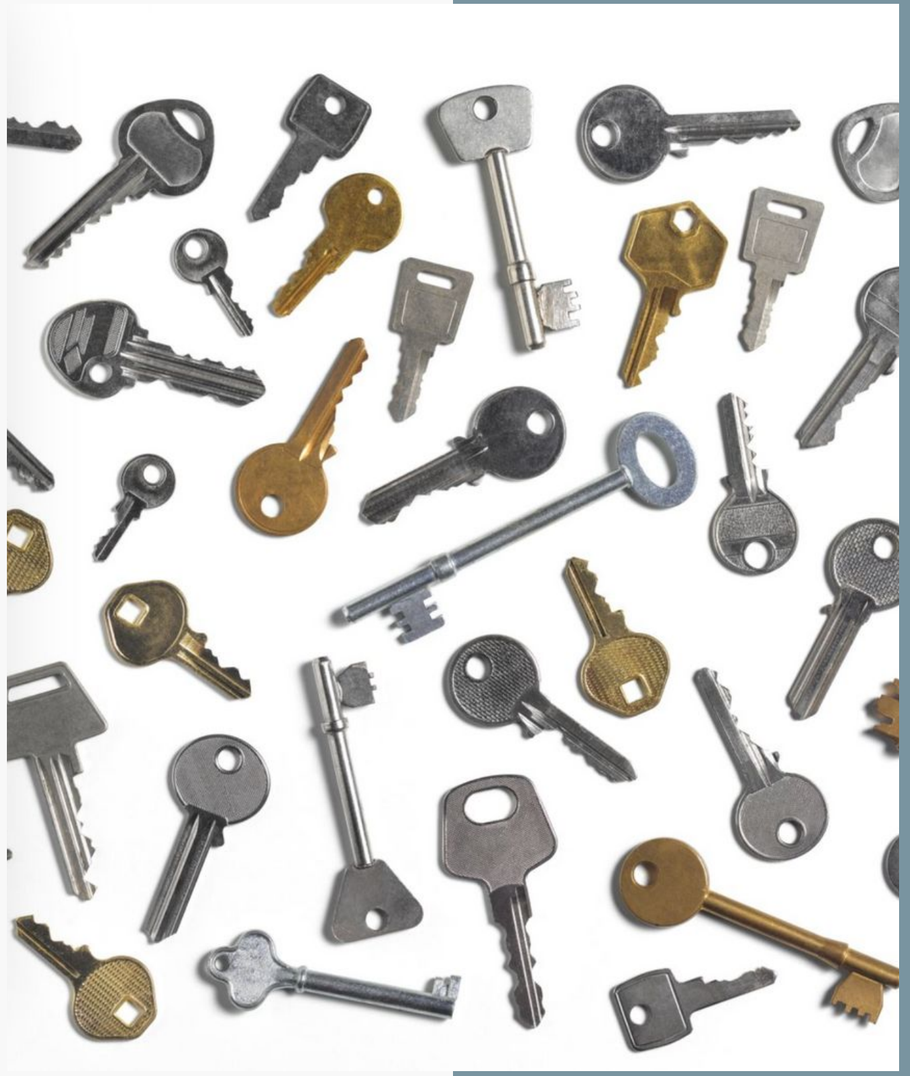
Travelling Thief Problem - Challenge

				GECCO	RKO			
Instance	N	M	BKS	OF	OF	RPDb	RPDa	best found at (s)
lin105_n104_uncorr_05	105	104	9676.76	9676.76	9676.76	0.00	0.00	3.09
lin105_n312_bounded-strongly-corr_02	105	312	24237.30	24227.32	24237.30	0.00	0.06	375.64
lin105_n520_uncorr_01	105	520	26578.58	26578.58	26578.58	0.00	0.00	7.83

Overview

Contents:

- Encoding and decoding with vectors of random keys
- Examples of encoding and decoding with random keys
- Random-Key Optimizers (RKO)
  - Components
  - Metaheuristics
  - Local Search
- RKO applications in discrete optimization problems
- API



Conclusion

Our extensive testing on NP-hard problems demonstrated RKO's **effectiveness** in producing optimal or near-optimal solutions efficiently. The **framework's flexibility** in incorporating diverse metaheuristics, coupled with its superior performance across different problem domains, underscores its **robustness** and **potential** as a **tool for discrete optimization**.

Thanks

A full version tech report is titled RANDOM-KEY OPTIMIZERS FOR COMBINATORIAL OPTIMIZATION, is available at

https://mauricio.resende.info/doc/RKO_ver_2024-11-02.pdf

antonio.chaves@unifesp.br mgcr@bekeley.edu rmas@cin.ufpe.br