

# Multiobjective Optimization

## Neighborhood Exploration

Igor Machado Coelho

20/09/2024-27/09/2024

- 1 Module: Neighborhood Exploration
- 2 Neighborhood Exploration
- 3 Neighborhood Exploration Primitives
- 4 Local Search and Refinement Heuristics
- 5 Advanced Topic: Multi Improvement
- 6 Practical Exercise
- 7 Discussions

## 8 Unused Slides

## 9 Agradecimentos

# Section 1

## Module: Neighborhood Exploration

# Requirements

The requirements for this class are:

- Data Structures and Algorithmic Complexity
  - Graph concepts
- Programming in Python or C/C++
- Module 1 - Fundamentals
- Module 2 - Greedy

# Topics



## Section 2

# Neighborhood Exploration

# Some Definitions (remember from Module 2)

Recall basic definitions for an optimization problem, such as solutions and evaluations, and classic NP-hard problems such as the Knapsack Problem and Traveling Salesman Problem. More precisely:

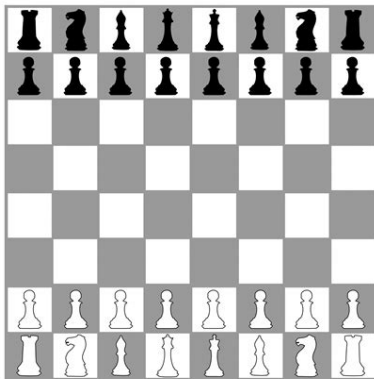
- The  $XS$  denotes a *Solution Space*, where  $XE$  is an *Evaluation Space* (or *Objective Space*)
  - The pair  $XES = \langle XS, XE \rangle$  denotes the *XESolution space*
  - A *SO optimization problem* is defined by the triple  $\langle XS, XE, f \rangle$
- The space  $XE$  can be partitioned into  $XE = XFeasible \cup XInfeasible$ , where  $XFeasible \cap XInfeasible = \emptyset$
- $XS$  denotes all *valid representations of a solution*, that are *structurally correct*
  - it may include *infeasible* solutions  $s$ , that are *valid*, but with *infeasible* evaluation  $f(s) \in XInfeasible$
  - it depends on how the problem is modeled, but it's not uncommon to have  $XInfeasible \neq \emptyset$
- The optimal solution  $s^*$  is always feasible  $f(s^*) \in XFeasible$ , unless the problem is *impossible*



# Neighborhood

Given a solution  $s \in XS$ , we define a *neighbor solution*  $s' \in XS$  as:

- a *neighborhood* (or *neighborhood structure*)  $\mathcal{N}(s)$  is a set of solutions reachable by some *move function/operator*  $m : XS \mapsto XS$
- we say that  $s' \in \mathcal{N}(s) \iff \exists m \text{ such that } s' = m(s)$ 
  - or, typically denoted by operator  $\oplus$  notation:  $s' = s \oplus m$



# Reachability of Solutions and Move Composition

We recall an instance of the Knapsack Problem with 5 items and consider solutions  $s_1 = (01001)$  and  $s_2 = (11010)$  from  $XS$

- We consider the following move definition  $M^{(I)} = \{m_1, m_2, \dots, m_i\} = \{\text{change the value of the bit } i\}$
- We can find moves  $m_1, m_4, m_5 \in M^{(I)}$  such that  $s_2 = ((s_1 \oplus m_1) \oplus m_4) \oplus m_5$ 
  - this changes the values of the first, fourth and fifth bits
  - the following intermediate solutions are visited in this path:  
 $s_1 = (01001) \rightarrow (11001) \rightarrow (11011) \rightarrow (11010) = s_2$
- Alternatively, a composite move  $m_{1,4,5}$  could be built with function composition:  $m_{1,4,5} = m_5 \circ m_4 \circ m_1$ ; or  $m_{1,4,5} = \bigcirc_{m \in (m_1, m_4, m_5)} m$  or by using  $\mapsto$  sequential notation:  $m_{1,4,5} = m_1 \mapsto m_4 \mapsto m_5$ 
  - In other words,  $m_{1,4,5}(s) = m_5(m_4(m_1(s)))$
- Now we consider moves  $M^{(II)}$  where two bits  $i$  and  $j$  are simultaneously changed (**Exercise:** What is the size of this neighborhood?)
  - Solution  $s_2$  could never be reachable by  $s_1$  in such neighborhood!

# Move Cost

Given a move  $m$  and function  $f$ , we can compute the *move cost*  $\bar{m}^f$  (or simply  $\bar{m}$ ) in the following way:

- given an evaluation function  $f : XS \mapsto XE$ , and  $e = f(s)$  denoting the *evaluation* of a solution  $s$  (when  $f$  is known, it can be omitted)
- given a *neighbor*  $s' = m(s)$  the *move cost*  $\bar{m}(s)$  is defined by  $\bar{m}(s) = \bar{m}^f(s) = f(s') - f(s)$
- naturally, any  $e \in XE$  space must support *add* and *subtract* basic arithmetics
- we **do not** require  $XE$  to be a *total order*, although this is true for *single objective optimization*, i.e., *minimization* or *maximization*
- we say that moves  $M = (m_1, m_2, \dots)$  are *independent* if *composite move*  $m' = \bigcirc_{m \in M} m$  has a fixed cost  $\bar{m}'(s) = \sum_{m \in M} \bar{m}(s)$ ,  $\forall s \in XS$ 
  - this is an important property for newer neighborhood strategies in literature!

# Example for the Traveling Salesman Problem (euclidean)

Let's think of a neighborhood structure for the TSP. What is a move? How much does it cost?



Figure 2: [https://simple.wikipedia.org/wiki/Travelling\\_salesman\\_problem](https://simple.wikipedia.org/wiki/Travelling_salesman_problem)

## Section 3

# Neighborhood Exploration Primitives

# Neighborhood Exploration: Basic Primitives

Given a solution  $s \in XS$ , a neighborhood  $\mathcal{N}$  and its associated *move set*  $\mathcal{M} = \{m_1, m_2, m_3, \dots\}$  such that  $\mathcal{N} = \{m_1(s), m_2(s), m_3(s), \dots\}$ , we define two basic neighborhood exploration primitives: `RandomMove` and `AllMoves`.

- the `RandomMove` returns a *random move*  $\{m\}$  from move set  $\mathcal{M}$
- the `AllMoves` returns a sequence  $\mathcal{M}^P = (m_1, m_2, \dots)$  of move set  $\mathcal{M}$

Typically, two complementary primitives are built on top of `AllMoves`: `FirstMove` and `NextMove`.

- $FirstMove(\mathcal{N}, s)$  returns  $\{m_1\}$ , where  $m_1$  is the first move from  $\mathcal{M}^P$
- $NextMove(\mathcal{N}, s, m_i)$  returns  $\{m_{i+1}\}$ , where  $m_i$  is a move from  $\mathcal{M}^P$

Note that returned moves may not exist, so one must check it! (typical *optional* behavior. . .)

# Neighborhood Exploration: Find Primitives

Given a neighborhood  $\mathcal{N}$  and solution  $s \in XS$ , we define three neighborhood exploration primitives: `FindAny`, `FindFirst` and `FindBest`.

- the `FindAny` tries to find *any* move  $m'$  with  $s' = m'(s) \in \mathcal{N}(s)$  that *improves*  $s$ 
  - we assume a more restricted neighborhood  $\mathcal{N}_{\leq \kappa} \subseteq \mathcal{N}$ , where  $|\mathcal{N}_{\leq \kappa}| \leq k$
  - assuming *minimization*, if such  $s' \in \mathcal{N}_{\leq \kappa}(s)$  exists, then  $f(s') < f(s)$
- the `FindFirst` tries to find *the first* move  $m_i$  with  $s_i = m_i(s) \in \mathcal{N}(s) = \{s_1, \dots, s_i, \dots\}$  that *improves* current solution  $s$ 
  - assuming *minimization*, if such  $s_i \in \mathcal{N}(s)$  exists, then  $i$  is the *smallest value* such that  $f(s_i) < f(s)$
- the `FindBest` tries to find *the best* move  $m^*$  with  $s^* = m^*(s) \in \mathcal{N}(s)$  that *improves* current solution  $s$ 
  - assuming *minimization*, if such  $s^* \in \mathcal{N}(s)$  exists, then  $f(s^*) < f(s)$  and  $f(s^*) \leq f(s')$ ,  $\forall s' \in \mathcal{N}(s)$

# Pseudocode for Find Primitives: FindAny

The `FindAny` considers, without loss of generality, a minimization function  $f$ , a neighborhood  $\mathcal{N}$ , a value  $\kappa_{max}$ , a pseudorandom function  $\xi(\cdot)$ , stop criteria  $stop(\cdot)$  and current solution  $s$ . It can be implemented in the following way:

```
procedure FINDANY( $f(\cdot)$ ,  $\mathcal{N}(\cdot)$ ,  $\kappa_{max}$ ,  $\xi(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )  
     $k \leftarrow 0$   
    while  $k < \kappa_{max}$  and not  $stop(time(), f(s))$  do  
         $m \leftarrow RandomMove(\mathcal{N}, s, \xi)$   
        if  $\bar{m}^f(s) < 0$  then  
            return  $\{m\}$   
        else  
             $k \leftarrow k + 1$   
        end if  
    end while  
    return  $\{\}$   
end procedure
```



# Pseudocode for Find Primitives: FindFirst

The `FindFirst` considers, without loss of generality, a minimization function  $f$ , a neighborhood  $\mathcal{N}$ , stop criteria  $stop(\cdot)$  and current solution  $s$ . It can be implemented in the following way:

```
procedure FINDFIRST( $f(\cdot)$ ,  $\mathcal{N}(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )  
     $m \leftarrow FirstMove(\mathcal{N}, s)$   
    while  $\exists m$  and not  $stop(time(), f(s))$  do  
        if  $\bar{m}^f(s) < 0$  then  
            return  $\{m\}$   
        else  
             $m \leftarrow NextMove(\mathcal{N}, s, m)$   
        end if  
    end while  
    return  $\{\}$   
end procedure
```

# Pseudocode for Find Primitives: FindBest

The `FindBest` considers, without loss of generality, a minimization function  $f$ , a neighborhood  $\mathcal{N}$ , stop criteria  $stop(\cdot)$  and current solution  $s$ . It can be implemented in the following way:

```

procedure FINDBEST( $f(\cdot)$ ,  $\mathcal{N}(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )
   $m \leftarrow FirstMove(\mathcal{N}, s)$ 
  if  $\nexists m$  then return  $\{\}$ 
   $\langle e^*, m^* \rangle \leftarrow \langle \bar{m}^f(s), m \rangle$ 
  while  $\exists m$  and not  $stop(time(), f(s))$  do
    if  $\bar{m}^f(s) < e^*$  then
       $\langle e^*, m^* \rangle \leftarrow \langle \bar{m}^f(s), m \rangle$ 
    end if
     $m \leftarrow NextMove(\mathcal{N}, s, m)$ 
  end while
  if  $e^* < 0$  then return  $\{m^*\}$  else return  $\{\}$ 
end procedure

```

## Section 4

# Local Search and Refinement Heuristics

# Heuristics for Neighborhood Exploration and Local Optima

Given a neighborhood  $\mathcal{N}$  and a solution  $s$ , we can explore it, in order to improve solution  $s$  by finding a better *neighbor*  $s'$

Some heuristics for neighborhood exploration are classic, mainly three: *random selection* (RS); *first improvement* (FI); and *best improvement* (BI). We have also proposed a *multi improvement* (MI) strategy that will be studied later.

These are also called *refinement heuristics* and are the foundations for several *local search* (LS) algorithms.

Differently from a *global search* (GS) algorithm, that tries to find an *optimal solution*, a *local search* tries to find a *locally optimal solution* regarding some specific neighborhood  $\mathcal{N}$ .

- So, recalling the basic definitions with  $XE$  as a *total order*, we define *local optima*  $s^* \in XS$ , given neighborhood  $\mathcal{N}$  and a solution  $s \in XS$ :
  - For *minimization*, we have that  $f(s^*) \leq f(s'), \forall s' \in \mathcal{N}(s)$
  - For *maximization*, we have that  $f(s^*) \geq f(s'), \forall s' \in \mathcal{N}(s)$

# Some Refinement Heuristics and Local Search

In the same way that *constructive heuristics* are able to generate initial solutions, the *Refinement Heuristics* can try to improve them. In special, *local search* algorithms are refinement heuristics that try to reach some *local optima* related to one or many neighborhoods.

We have seen some neighborhood exploration primitives that try to generate an *improving move* over  $\mathcal{N}$  for solution  $s \in XS$ . Now, the refinement heuristic  $\mathcal{H}$  will try to return an *improving neighbor*  $s' = \mathcal{H}(s) \in XS$  (naturally, if  $\exists s'$  then  $f(s') < f(s)$  for *minimization*).

We begin with classic refinement heuristics: *Random Selection*, *First Improvement* and *Best Improvement*.

# Refinement Heuristic: Random Selection

Given a neighborhood  $\mathcal{N}$  and a parameter  $\kappa_{max}$ , the *Random Selection* (RS) heuristic is an implementation of the primitive FindAny:

- RS tries to find *any* solution  $s' \in \mathcal{N}(s)$  that *improves* current solution  $s \in XS$
- RS is limited to  $k_{max}$  tries

```
procedure RANDOMSELECTION( $f(\cdot)$ ,  $\mathcal{N}(\cdot)$ ,  $\kappa_{max}$ ,  $\xi(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )  
   $m \leftarrow \text{FindAny}(f, \mathcal{N}, \kappa_{max}, \xi(\cdot), stop, s)$   
  if  $\exists m$  then  
    return  $\{m(s)\}$   
  else  
    return  $\{\}$   
  end if  
end procedure
```

# Refinement Heuristic: First Improvement

Given a neighborhood  $\mathcal{N}$ , the *First Improvement* (FI) heuristic is an implementation of the primitive FindFirst:

- FI tries to find *the first* solution  $s' \in \mathcal{N}(s)$  that *improves* current solution  $s \in XS$

**procedure** FIRSTIMPROVEMENT( $f(\cdot)$ ,  $\mathcal{N}(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )

$m \leftarrow \text{FindFirst}(f, \mathcal{N}, stop, s)$

**if**  $\exists m$  **then**

**return**  $\{m(s)\}$

**else**

**return**  $\{\}$

**end if**

**end procedure**

# Refinement Heuristic: Best Improvement

Given a neighborhood  $\mathcal{N}$ , the *Best Improvement* (BI) heuristic is an implementation of the primitive FindBest:

- FI tries to find *the best* solution  $s' \in \mathcal{N}(s)$  that *improves* current solution  $s \in XS$

**procedure** BESTIMPROVEMENT( $f(\cdot)$ ,  $\mathcal{N}(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )

$m \leftarrow \text{FindBest}(f, \mathcal{N}, stop, s)$

**if**  $\exists m$  **then**

**return**  $\{m(s)\}$

**else**

**return**  $\{\}$

**end if**

**end procedure**



# Classic Local Search techniques

We now explore some classic local search techniques, such as: *Hill Climbing* (HC), *Random Descent Method* (RDM) and *Variable Neighborhood Descent* (VND).

## Combining refinement heuristics

Each of these local search methods can be combined with all previous refinement heuristics.

## Reaching local optimality

Not all of these local search methods can guarantee local optimality, but *they will try!*.

# Local Search: Hill Climbing

Given a refinement heuristic  $\mathcal{H}$  that explores some neighborhood and a solution  $s \in XS$ , the *Hill Climbing* (HC) is an iterative algorithm that finds a *local optimum*. HC is very simple and popular (see wiki).

- HC is also known as *Simple Hill Climbing*, when integrated with *FI*
- HC is also known as *Steepest Ascent/Descent Hill Climbing*, when integrated with *BI*
- HC is also known as *Stochastic Hill Climbing*, when integrated with *RS*

**procedure** HILLCLIMBING( $f(\cdot)$ ,  $\mathcal{H}(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )

$s' \leftarrow \mathcal{H}^{f, stop}(s)$

**while**  $\exists s'$  **and not**  $stop(time(), f(s'))$  **do**

$s \leftarrow s'$

$s' \leftarrow \mathcal{H}^{f, stop}(s)$

**end while**

**return**  $\{s\}$

**end procedure**

# Local Search: Random Descent Method

Given a neighborhood  $\mathcal{N}$ , a parameter  $\kappa_{max}$  and a solution  $s \in XS$ , the *Random Descent Method* (RDM) is an iterative algorithm that tries to find a *local optimum*. It is very similar to the *Stochastic Hill Climbing*.

```
procedure RANDOMDESCENTMETHOD( $f(\cdot)$ ,  $\mathcal{N}(\cdot)$ ,  $\kappa_{max}$ ,  $\xi$ ,  $stop(\cdot)$ ,  $s$ )  
   $k \leftarrow 0$   
  while  $k < \kappa_{max}$  and not  $stop(time(), f(s))$  do  
     $m \leftarrow RandomMove(\mathcal{N}, s, \xi)$   
    if  $\exists m$  and  $\bar{m}^f(s) < 0$  then  
       $s \leftarrow m(s)$   
       $k \leftarrow 0$   
    else  
       $k \leftarrow k + 1$   
    end if  
  end while  
  return  $\{s\}$   
end procedure
```

# Local Search: Variable Neighborhood Descent

Given *multiple* refinement heuristics  $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_k$  that explore some neighborhoods and a solution  $s \in XS$ , the *Variable Neighborhood Descent* (VND) is an iterative algorithm that finds a *local optimum* regarding *all neighborhoods*.

- VND is very sensitive the order of neighborhoods
  - typically, start from *smaller* then explore the *larger*

```
procedure VND( $f(\cdot)$ ,  $\mathcal{H}_k(\cdot)$ ,  $stop(\cdot)$ ,  $s$ )  
   $i \leftarrow 1$   
  while  $i \leq k$  and not  $stop(time(), f(s))$  do  
     $s' \leftarrow \mathcal{H}_i^{f, stop}(s)$   
    if  $\exists s'$  then  $\langle s, i \rangle \leftarrow \langle s', 1 \rangle$  else  $i \leftarrow i + 1$   
  end while  
  return  $\{s\}$   
end procedure
```

# Local Search: Randomized Variable Neighborhood Descent

In 2010, our research group proposed a *randomized* version of VND, called *Randomized Variable Neighborhood Descent* (RVND).

In fact, two subgroups independently proposed the same technique (see Souza 2010 and Anand 2010)

- RVND is not sensitive the order of neighborhoods

```
procedure RVND( $f(\cdot)$ ,  $\mathcal{H}'_k(\cdot)$ ,  $\xi$ ,  $stop(\cdot)$ ,  $s$ )  
   $\mathcal{H} \leftarrow shuffle^{\xi}(\mathcal{H}')$   
   $i \leftarrow 1$   
  while  $i \leq k$  and not  $stop(time(), f(s))$  do  
     $s' \leftarrow \mathcal{H}_i^{f, stop}(s)$   
    if  $\exists s'$  then  $\langle s, i \rangle \leftarrow \langle s', 1 \rangle$  else  $i \leftarrow i + 1$   
  end while  
  return  $\{s\}$   
end procedure
```

## Section 5

### Advanced Topic: Multi Improvement

# Multimprovement: the Idea

Given a solution  $s \in XS$ , a neighborhood  $\mathcal{N}$  and its associated *move set*  $\mathcal{M}$ , the *Multi Improvement* (MI) heuristic is an implementation of the primitive FindFirst or FindBest over a compound neighborhood  $\mathcal{N}^\circ$ .

The compound neighborhood  $\mathcal{N}^\circ$  is associated to a compound move set  $\mathcal{M}^\circ = \{m^\circ | m^\circ = \bigcirc_{m \in \mathcal{X}} m, \forall \mathcal{X} \in {}^\pi \mathcal{M}^*\}$  that be seen as a set of *all move compositions* for  $\mathcal{M}^*$ , which is a *subset of the powerset*  $2^{\mathcal{M}}$  only containing *independent moves* for  $s$ . Note that operator  $\in {}^\pi$  takes a set of the powerset and also performs a permutation, transforming the selected set into a sequence.

Finding a “best” compound move can only be done exactly (and it’s even NP-hard for some neighborhoods!). So finding a “first” solution can be feasible on practice, by employing some “greedy” strategy. In this sense, using CPU-GPU hybrid architecture can help deciding how such “FindFirst” operation can work efficiently, by organizing GPU blocks and shared memory in a smart way.

# Some formulation

Given  $s \in XS$ , a neighborhood  $\mathcal{N}$  and its *move set*  $\mathcal{M}$ , we can formulate this problem as the following *maximization* problem:

$$\max \bar{m}^\circ(s)$$

$$\mathcal{X} \in \mathcal{M}^* \subseteq 2^{\mathcal{M}}$$

$$m^\circ = \bigcirc_{m \in \mathcal{X}} m$$

Move Independence:

$$\bar{m}^\circ(s) = \sum_{m \in \mathcal{P}} \bar{m}(s), \forall \mathcal{P} \in \Pi(\mathcal{X})$$

Where,  $\Pi(C)$  is the set of all permutations of set  $C$ .



# Exploring the Multi Improvement technique

Please read recent articles from our research group!

- A benchmark on multi improvement neighborhood search strategies in CPU/GPU systems (2016)
- Exploring parallel multi-GPU local search strategies in a metaheuristic framework (2018)
- A DVND local search implemented on a dataflow architecture for the minimum latency problem (2018)
- Multi Improvement: uma Solução Alternativa para o Problema de Roteamento de Veículos (2019)
- A multi-improvement local search using dataflow and GPU to solve the minimum latency problem (2020)
- Finding the maximum multi improvement on neighborhood exploration (2022)

## Section 6

### Practical Exercise

# Implementing a Local Search (Step 1/3)

- Choose a language: Python or C/C++
- Consider the following data for a Knapsack Problem with  $n = 5$  items and capacity  $Q = 10$

5

10

1 1 1 5 5

1 2 3 7 8

- Save it into a file and read it
  - First load the  $n$  and  $Q$
  - Then, for each item, load each profit  $p_i$  and weight  $w_i$

# Implementing a Local Search (Step 2/3)

- Model the solution representation as an array (or list) of booleans or binary numbers
- Create a neighborhood structure and two neighborhood exploration techniques (example: best improvement and first improvement)
- Generate multiple initial solutions with some randomness (example, 1000)
- Compute the Average cost and Computational time taken for each of the two refinement heuristics
- Which of these are the best one?

# Implementing a Local Search (Step 3/3)

- Now, choose some Local Search technique, such as Hill Climbing (for BI, FI or RS) or RDM
- Generate multiple initial solutions with some randomness (example, 1000)
- Apply each of the two Local Search on them, for each generated solution
- Compute the Average cost and Computational time taken for each of the two local searches
- Generate bigger instances, to make the problem harder!
- Which one is better?

## Section 7

### Discussions

# Short discussion

## Current scenario: optimization problems in the university and work

- Do you know of any optimization problem that needs to be solved in the university or your work?
- Can exact methods solve them? Do you need heuristic methods?
- Read the introduction material from prof Marccone (in Portuguese): <http://www.decom.ufop.br/prof/marccone/Disciplinas/InteligenciaComput>

## Section 8

### Unused Slides



# Moves: Basic Primitives

Given a move  $m$ , a solution  $s \in XS$  and its evaluation  $e = f(s) \in XE$ , we define three basic move primitives: CanApply, Apply and Cost.

- the CanApply returns *true* only if  $m(s) \in XS$ , i.e., if the generated neighbor is a *valid solution* in  $XS$  space
  - this can be useful when moves are clearly defined, such as changing a bit  $i$  in a knapsack problem, but not all moves lead to valid solutions, for example, if knapsack capacity would be exceeded after move and that is not allowed in  $XS$
- the Apply primitive returns pair  $\langle m(s), m' \rangle$ , where  $m'$  is an *undo move*, such that,  $s = m'(m(s))$ 
  - only defined if CanApply is *true*
- the Cost primitive returns evaluation difference value
$$e_{diff} = f(m(s)) - f(s)$$
  - only defined if CanApply is *true*

A fourth non-basic primitive typically used is the ApplyUpdate, that returns both the *solution neighbor* and its *evaluation* in a pair  $\langle m(s), f(m(s)) \rangle$ .

# Neighborhood Exploration: FindNext Primitive (extra)

Although not commonly used, one can define a FindNext primitive:

- the FindNext tries to find *the next*  $s_i \in \mathcal{N}(s) = \{s_j, \dots, s_i, \dots\}$  that *improves* current solution  $s$ 
  - assuming *minimization*, if such  $s_i \in \mathcal{N}(s)$  exists, then  $i$  is the *smallest value* such that  $f(s_i) < f(s)$  and  $i > j$

## Section 9

# Agradecimentos

# Pessoas

Em especial, agradeço aos colegas que elaboraram bons materiais, como o prof. Raphael Machado, Kowada e Viterbo cujos conceitos formam o cerne desses slides.

Estendo os agradecimentos aos demais colegas que colaboraram com a elaboração do material do curso de Pesquisa Operacional, que abriu caminho para verificação prática dessa tecnologia de slides.

# Software

Esse material de curso só é possível graças aos inúmeros projetos de código-aberto que são necessários a ele, incluindo:

- pandoc
- LaTeX
- GNU/Linux
- git
- markdown-preview-enhanced (github)
- visual studio code
- atom
- revealjs
- groomit-mpx (screen drawing tool)
- xournal (screen drawing tool)
- ...

# Empresas

Agradecimento especial a empresas que suportam projetos livres envolvidos nesse curso:

- github
- gitlab
- microsoft
- google
- ...

# Reprodução do material

Esses slides foram escritos utilizando pandoc, segundo o tutorial ilectures:

- <https://igormcoelho.github.io/ilectures-pandoc/>

Exceto expressamente mencionado (com as devidas ressalvas ao material cedido por colegas), a licença será Creative Commons.

**Licença:** CC-BY 4.0 2020

Igor Machado Coelho

# This Slide Is Intentionally Blank (for goomit-mpx)