Construction-Based Metaheuristics

Leandro Montero

UCO Angers

2025-2026

based on lectures of Fabien Tricoire

Outline

Introduction

So far, we learned about (meta)heuristics for:

- Constructing a solution
- Improving a solution
  - Using local search
  - Mimicking natural behaviour

Introduction

So far, we learned about (meta)heuristics for:

- Constructing a solution
- Improving a solution
  - Using local search
  - Mimicking natural behaviour

It is also possible to iterate construction heuristics

- To improve a partial solution: (A)LNS
- To construct many good solutions: GRASP
- Actually, we already used a similar concept; when?

Introduction

Today, we will:

- Learn about a few construction-based metaheuristics
- Not go into every detail
- (for instance, no quantitative analysis)
- Learn a few more tricks that will be useful in the future

Outline

Greedy Randomised Adaptive Search Procedure

Greedy Randomised Adaptive Search Procedure (GRASP) aims at improving the behaviour of greedy construction algorithms.

Greedy Randomised Adaptive Search Procedure

Greedy Randomised Adaptive Search Procedure (GRASP) aims at improving the behaviour of greedy construction algorithms.

- Greedy construction provides little diversification
  $\rightarrow$ inject randomness in the process
- Too much randomness leads to nothing interesting
  $\rightarrow$ limit randomness

Greedy Randomised Adaptive Search Procedure

Greedy Randomised Adaptive Search Procedure (GRASP) aims at improving the behaviour of greedy construction algorithms.

- Greedy construction provides little diversification
  $\rightarrow$ inject randomness in the process
- Too much randomness leads to nothing interesting
  $\rightarrow$ limit randomness
- Constructed solutions can usually be improved quickly
  $\rightarrow$ perform local search

(in the following we assume that we are working on a minimisation problem)

GRASP: abstract algorithm

GRASP()

$x^* \leftarrow nil$
while stopping criterion not reached **do**
    $x \leftarrow randomisedConstruction()$
    $x' \leftarrow localSearch(x)$
    if $x^* = nil \vee z(x') < z(x^*)$ **then**
        $x^* \leftarrow x'$
    end if
end while
Return x^*

GRASP: abstract algorithm

GRASP()

$x^* \leftarrow nil$
while stopping criterion not reached **do**
  $x \leftarrow$ *randomisedConstruction*()
  $x' \leftarrow$ *localSearch*(x)
  if $x^* = nil \vee z(x') < z(x^*)$ **then**
    $x^* \leftarrow x'$
  end if
end while
Return x^*

- Things that we should specify
- The rest you already know...

GRASP: randomised construction

Traditionally in greedy construction we insert the solution component with best heuristic value.

How it is done in GRASP:

- Consider the restricted candidate list (RCL), containing the candidates with best heuristic values
- Randomly pick one of them and add it to the solution

GRASP: randomised construction

Traditionally in greedy construction we insert the solution component with best heuristic value.

How it is done in GRASP:

- Consider the restricted candidate list (RCL), containing the candidates with best heuristic values
- Randomly pick one of them and add it to the solution

Two possible traditional ways to compute RCL:

1. Cardinality restriction: select the k best candidates
2. Value restriction: accept until a certain threshold
   - Let $g(c)$ be the heuristic value for candidate c
   - g_{min} and g_{max} be the best and worst heuristic values among all candidates
   - Then RCL only contains elements l for which $g(l) \leqslant g_{min} + \alpha(g_{max} - g_{min})$

Constructing the RCL

Lists as data structures

- () is the empty list
- $\mathit{append}(L, e)$ returns a list equal to L + element e appended
- $\mathit{last}(L)$ returns the first element in L
- $\mathit{first}(L)$ returns the first element in L
- $\mathit{rest}(L)$ returns the rest of list L (i.e. every element in L except for $\mathit{first}(L)$)

In the following we assume that the list Λ of all candidates is always ordered by increasing heuristic values.

Constructing the RCL: version 1

Exercise: write the abstract algorithm for *generateRCL*$1(\Lambda, k)$, which returns a list of the k best candidates from Λ

Constructing the RCL: version 2

Exercise: write the abstract algorithm for *generateRCL*2(Λ, α), which returns a list of all candidates l from Λ such that $g(l) \leqslant g_{min} + \alpha(g_{max} - g_{min})$

Randomised construction

What to do with the RCL

- Greedy algorithms always select the best candidate
- GRASP always selects <u>one candidate from the RCL</u> randomly

Outline

Large Neighbourhood Search (LNS)

The reason why it is called Large Neighbourhood Search

The neighbourhood used in LNS is defined as all the possible ways to partially destroy a solution then repair it (aka ruin and recreate)

Large Neighbourhood Search (LNS)

The reason why it is called Large Neighbourhood Search

The neighbourhood used in LNS is defined as all the possible ways to partially destroy a solution then repair it (aka ruin and recreate)

Example: the TSP

- Destroy: remove some cities from the solution
- Repair: reinsert these cities in the solution

Issues

- The neighbourhood is too large to allow exhaustive exploration
- We are only interested in good neighbours anyway

LNS: neighbourhood exploration

Addressing this issue

Focus on good neighbours: use heuristics.

- To destroy the solution: destroy heuristics
- To repair the solution: repair (aka construction) heuristics
- It is actually hard to devise good destroy heuristics!

LNS: neighbourhood exploration

Addressing this issue

Focus on good neighbours: use heuristics.

- To destroy the solution: destroy heuristics
- To repair the solution: repair (aka construction) heuristics
- It is actually hard to devise good destroy heuristics!

Another issue

- There is little interest in repeatedly destroying and repairing a solution in the exact same way

LNS: neighbourhood exploration

Exercise: how can we destroy and repair several times with different results?

Quantity to destroy

We don't want to destroy too much

- We want to improve the current solution, not construct a new one from scratch
- We want to keep good features of the solution

Quantity to destroy

We don't want to destroy too much

- We want to improve the current solution, not construct a new one from scratch
- We want to keep good features of the solution

We don't want to destroy too little

- The more we destroy, the more opportunities for improvement

Quantity to destroy

We don't want to destroy too much

- We want to improve the current solution, not construct a new one from scratch
- We want to keep good features of the solution

We don't want to destroy too little

- The more we destroy, the more opportunities for improvement

We want flexibility

- No real need to always destroy the same amount
- Destroying small quantities leads to quicker iterations
- Destroying large quantities allows more diversification

How it's done: destroy between 1 and $k\%$ of the solution

Using several heuristics

Many meaningful ways to destroy a solution

- Remove the worst parts (greedy)
- Remove related parts
- Remove random parts (diversification)
- Anything problem-specific

Using several heuristics

Many meaningful ways to destroy a solution

- Remove the worst parts (greedy)
- Remove related parts
- Remove random parts (diversification)
- Anything problem-specific

The same holds for repairing

- Greedy construction
- Insert related parts
- More advanced than greedy (e.g. regrets)
- Anything problem-specific

A new pair of heuristics is randomly selected at each iteration

Using stochastic heuristics

We already talked about two ways to randomise construction heuristics:

- Use a restricted candidate list (GRASP)
- Compute weights then use a roulette wheel for selection (ACO)

Using stochastic heuristics

We already talked about two ways to randomise construction heuristics:

- Use a restricted candidate list (GRASP)
- Compute weights then use a roulette wheel for selection (ACO)

There exists another way: adding noise

Adding noise to a heuristic

- General idea: modify the heuristic values with randomness
- Let C be the heuristic value of a move
- One possibility: consider $C + \delta$ instead, where δ is a random number from $[-\eta, \eta]$ (η is a parameter)
- Another possibility: consider αC instead, where α is a random number close to 1.

LNS: abstract algorithm

LNS()

$x \leftarrow \mathit{construction}()$
$x^* \leftarrow x$
while stopping criterion not reached **do**
    $d \leftarrow \mathit{selectDestroy}()$
    $r \leftarrow \mathit{selectRepair}()$
    $x' \leftarrow r(d(x))$
    $x \leftarrow \mathit{accept}(x, x')$
    if $z(x) < z(x^*)$ **then**
        $x^* \leftarrow x$
    end if
end while
Return x^*

LNS: Advanced neighbourhood selection

Key idea: heuristics are not equally useful

Mechanism #1: give more importance to some heuristics

- Each heuristic has a certain weight
- Use roulette wheel to select heuristics at each iteration

LNS: Advanced neighbourhood selection

Key idea: heuristics are not equally useful

Mechanism #1: give more importance to some heuristics

- Each heuristic has a certain weight
- Use roulette wheel to select heuristics at each iteration

Mechanism #2: adaptiveness

Use the search history to determine which heuristics should have more importance.

- Keep track of the success rate of each heuristic using scores
- Divide the search into time segments (e.g. 100 iterations)
- Between segments, update the weights based on:
  - scores in the last segment
  - current values of weights (reflecting all scores since the beginning of the search)

Additional resources

- A general heuristic for vehicle routing problems (ALNS)
- Adaptive memory programming: a unified view of metaheuristics