Newer Metaheuristics Based on Local Search

Leandro Montero

UCO Angers

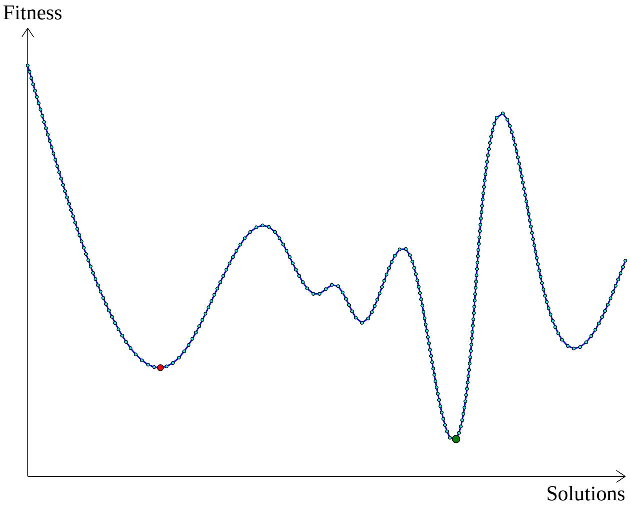
2025-2026

based on lectures of Fabien Tricoire

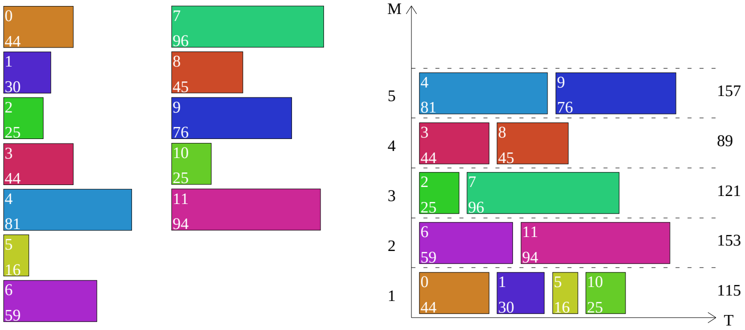
Outline

Outline

Never forget this!!



Remember this local optimum



- Tabu Search overcomes this situation with local degradations and short-term memory
- But there are other possibilities...
  (still based on Local Search)

Avoiding local optima with Local Search methods

Given a neighbourhood/operator, we want to escape one of its local optima

Exercise: suggest other possibilities

Avoiding local optima with Local Search methods

Given a neighbourhood/operator, we want to escape one of its local optima

Exercise: suggest other possibilities

Possibilities include:

1. Introduce perturbation
   - We will see this later (ILS, VNS)
2. Change the neighbourhood
   - Let's discuss this now!

Outline

Back to definitions

Definition

Neighbourhood: the set of solutions that can be produced by applying a given operator to a given solution.

Back to definitions

Definition

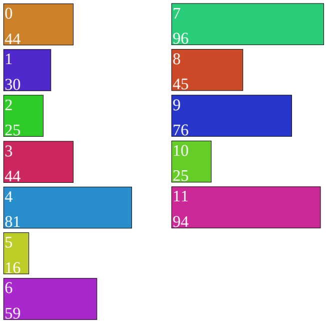
Neighbourhood: the set of solutions that can be produced by applying a given operator to a given solution.

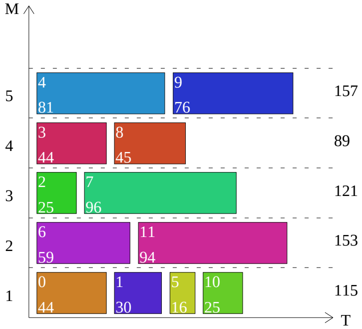
So we can either:

- Change the given (incumbent) solution (equivalent to perturbation)
- Change the given neighbourhood operator (aka neighbourhood from now on)

Variable Neighbourhood Descent (VND) aims at escaping local optima by changing neighbourhoods when appropriate, and picking the best solution in the current neighbourhood.

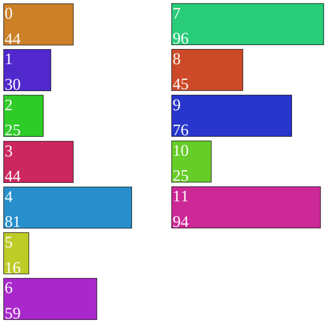
Changing neighbourhoods

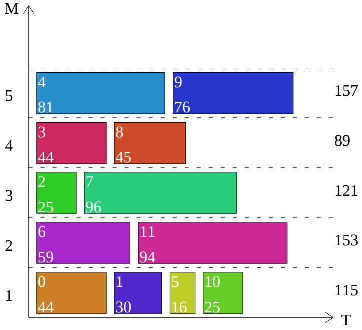




Exercise: find a neighbourhood for which this solution is not locally optimal

Changing neighbourhoods





Exercise: find a neighbourhood for which this solution is not locally optimal

Once more, several possibilities, including:

- Swap two jobs
- Move more than one job

For instance, here we could swap jobs 3 and 4 and improve the objective (what would be the new objective value?)

Exercise: write *bestSwapNeighbour*(x)

It returns the best neighbour of x using the *swap* neighbourhood

bestSwapNeighbour(x)

Exercise: write *bestSwapNeighbour*(x)

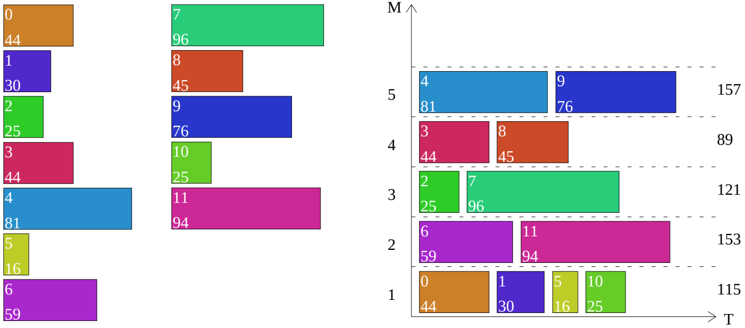
It returns the best neighbour of x using the *swap* neighbourhood

bestSwapNeighbour(x)

```
bestDuration ← ∞
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        if x[i] ≠ x[j] then
            x[i] ↔ x[j]
            if getDuration(x) < bestDuration then
                bestDuration ← getDuration(x)
                bestJob1 ← i
                bestJob2 ← j
            end if
            x[i] ↔ x[j]
        end if
    end for
end for
x′ ← x
x′[bestJob1] ↔ x′[bestJob2]
Return x′
```

Steepest descent on the swap neighbourhood

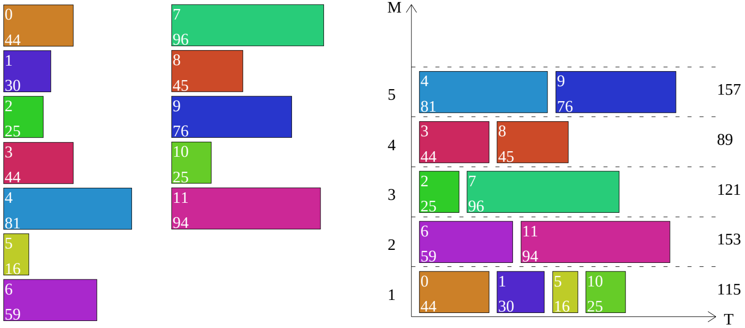
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	1	1	3	4	5	1	2	3	4	5	1	2

Steepest descent on the swap neighbourhood

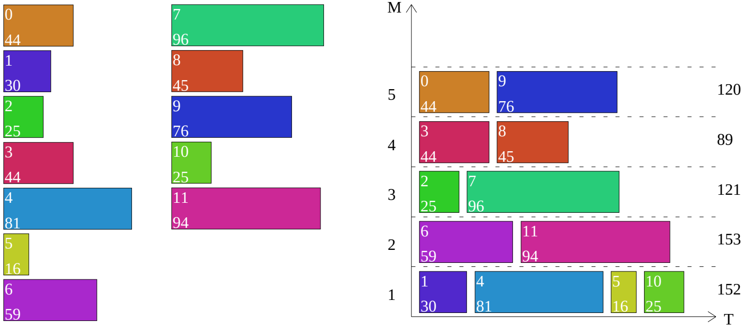
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	1	1	3	4	5	1	2	3	4	5	1	2

Steepest descent on the swap neighbourhood

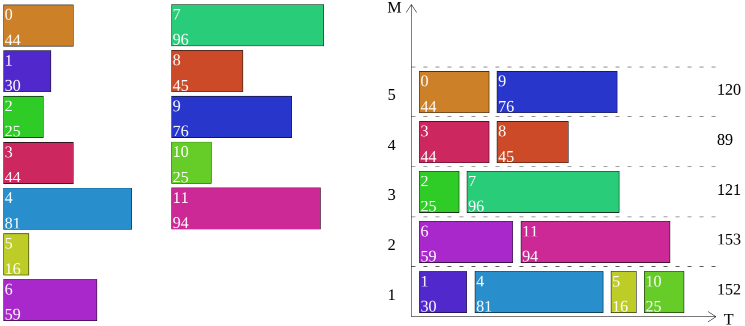
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	5	1	3	4	1	1	2	3	4	5	1	2

Steepest descent on the swap neighbourhood

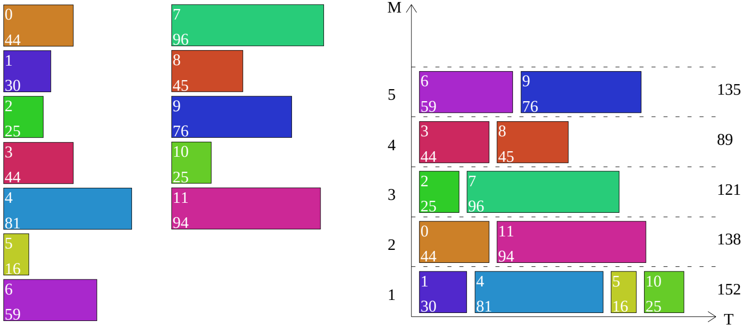
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	5	1	3	4	1	1	2	3	4	5	1	2

Steepest descent on the swap neighbourhood

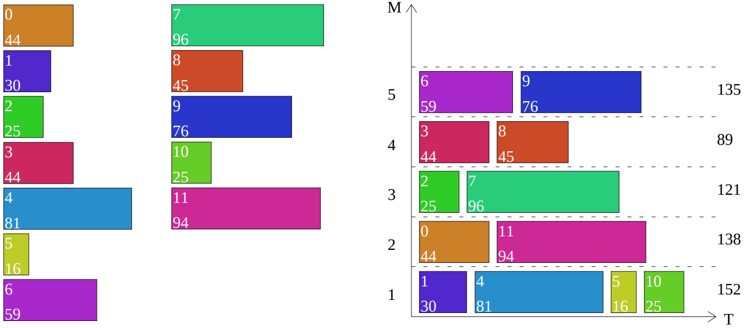
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	2	1	3	4	1	1	5	3	4	5	1	2

Steepest descent on the swap neighbourhood

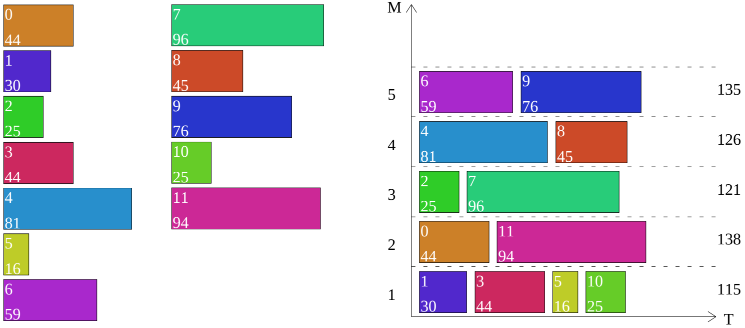
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	2	1	3	4	1	1	5	3	4	5	1	2

Steepest descent on the swap neighbourhood

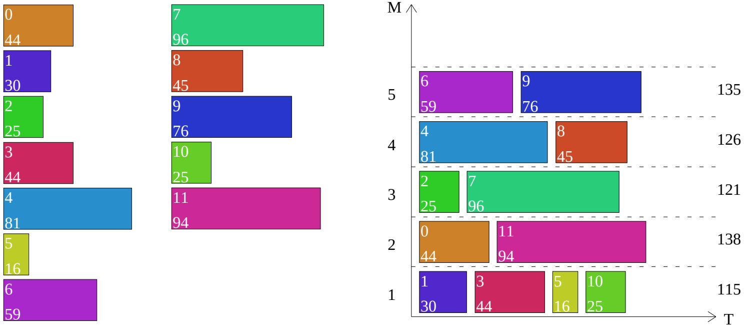
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	2	1	3	1	4	1	5	3	4	5	1	2

Steepest descent on the swap neighbourhood

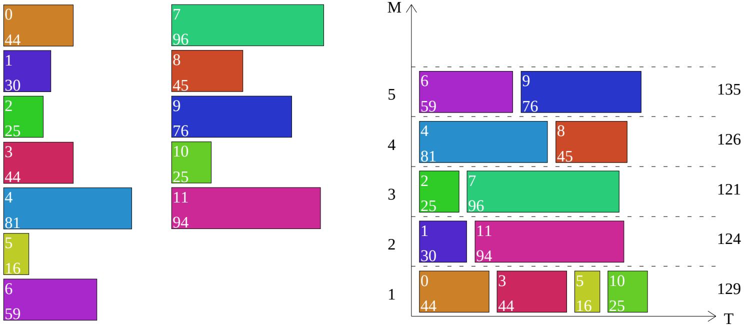
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	2	1	3	1	4	1	5	3	4	5	1	2

Steepest descent on the swap neighbourhood

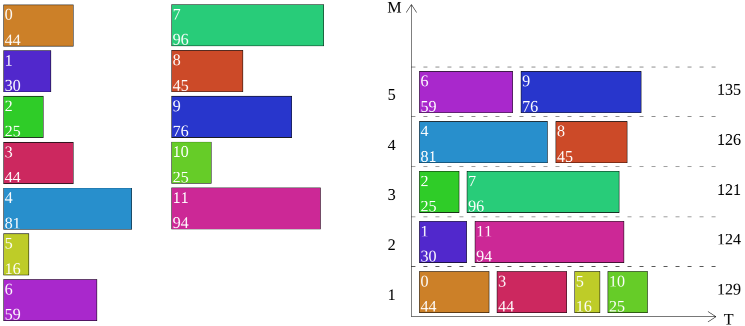
Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	1	2	3	1	4	1	5	3	4	5	1	2

Steepest descent on the swap neighbourhood

Exercise: perform a steepest descent with the swap neighbourhood



Job	0	1	2	3	4	5	6	7	8	9	10	11
Machine	1	2	3	1	4	1	5	3	4	5	1	2

Optimum found!

Choosing neighbourhoods

The facts:

- Local optimum x for neighbourhood N_1 might be not locally optimal with N_2
- In such a case, using N_2 helps escaping

Choosing neighbourhoods

The facts:

- Local optimum x for neighbourhood N_1 might be not locally optimal with N_2
- In such a case, using N_2 helps escaping

This leads to more questions though:

1. Given N_1, how do we find/design N_2?
2. Why not use N_2 directly?
3. What do we do when x is locally optimal with N_2?
4. What do we do when N_2 allowed to escape from a local optimum for N_1?

Question 1: given N_1, how do we design N_2?

- We know that: $z(x) \leqslant min\{z(t) | t \in N_1(x)\}$

Question 1: given N_1, how do we design N_2?

- We know that: $z(x) \leqslant min\{z(t)|t \in N_1(x)\}$
- We want that: $z(x) \geqslant min\{z(t)|t \in N_2(x)\}$

Question 1: given N_1, how do we design N_2?

- We know that: $z(x) \leqslant min\{z(t)|t \in N_1(x)\}$
- We want that: $z(x) \geqslant min\{z(t)|t \in N_2(x)\}$

There are (at least) 3 ways to comprehend this:

- N_2 should be as different as possible from N_1
- N_1 should be nested in N_2
- N_2 should be "larger" than N_1, and allow more operations of the same kind, or larger operations

Question 1: given N_1, how do we design N_2?

- We know that: $z(x) \leqslant min\{z(t) | t \in N_1(x)\}$
- We want that: $z(x) \geqslant min\{z(t) | t \in N_2(x)\}$

There are (at least) 3 ways to comprehend this:

- N_2 should be as different as possible from N_1
- N_1 should be nested in N_2
- N_2 should be "larger" than N_1, and allow more operations of the same kind, or larger operations

<u>Exercise: in which category falls our case?</u>

- N_1: move a job
- N_2: swap two jobs

Question 1: given N_1, how do we design N_2?

- We know that: $z(x) \leqslant min\{z(t) | t \in N_1(x)\}$
- We want that: $z(x) \geqslant min\{z(t) | t \in N_2(x)\}$

There are (at least) 3 ways to comprehend this:

- N_2 should be as different as possible from N_1
- N_1 should be nested in N_2
- N_2 should be "larger" than N_1, and allow more operations of the same kind, or larger operations

<u>Exercise: in which category falls our case?</u>

- N_1: move a job
- N_2: swap two jobs

- Swapping two jobs can be achieved by applying N_1 twice
- Moves from N_1 cannot be reproduced using N_2

Question 2: why not use N_2 directly?

Exercise: why can it be better to use N_1 instead of N_2?

Question 2: why not use N_2 directly?

Exercise: why can it be better to use N_1 instead of N_2?

If N_2 is likely to "unlock" from N_1, it is also likely to be a larger neighbourhood, longer to explore

Exercise: what are the complexities of Move and Swap?

Question 2: why not use N_2 directly?

Exercise: why can it be better to use N_1 instead of N_2?

If N_2 is likely to “unlock” from N_1, it is also likely to be a larger neighbourhood, longer to explore

Exercise: what are the complexities of Move and Swap?

- Move: $O(n \cdot m)$
- Swap: $O(n^2)$
- (Keep in mind that $n > m$)

Question 2: why not use N_2 directly?

Exercise: why can it be better to use N_1 instead of N_2?

If N_2 is likely to "unlock" from N_1, it is also likely to be a larger neighbourhood, longer to explore

Exercise: what are the complexities of Move and Swap?

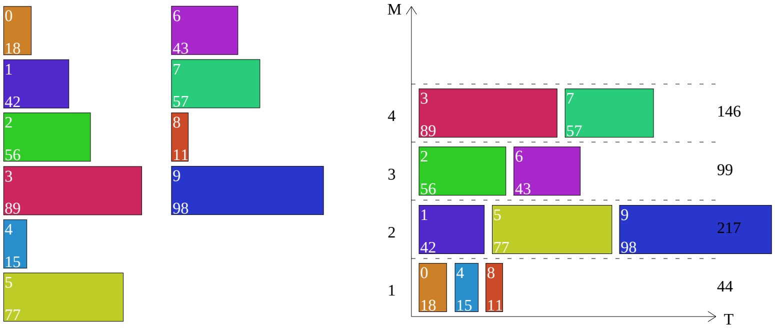
- Move: $O(n \cdot m)$
- Swap: $O(n^2)$
- (Keep in mind that $n > m$)

This leads us to a key idea in Variable Neighbourhood Descent (VND):

- Use "small, simple" neighbourhoods first
- Use larger, more powerful neighbourhoods to escape from local optima for smaller neighbourhoods

Question 3: what to do when x is locally optimal with N_2?

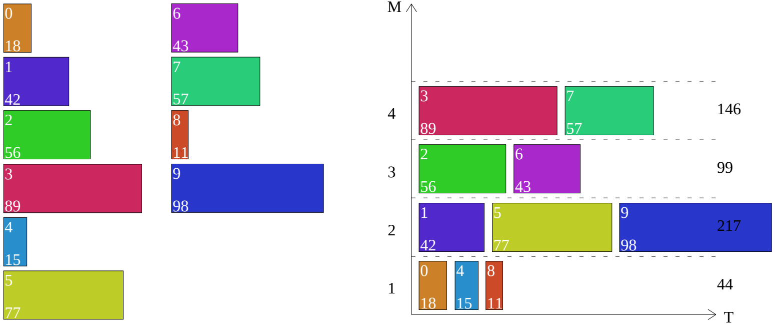
Exercise: improve this solution using previously mentioned ideas



Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	2	3	4	1	2

Question 3: what to do when x is locally optimal with N_2?

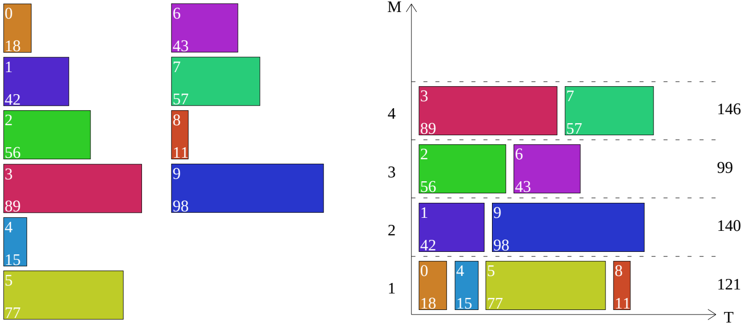
Exercise: improve this solution using previously mentioned ideas



Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	2	3	4	1	2

Question 3: what to do when x is locally optimal with N_2?

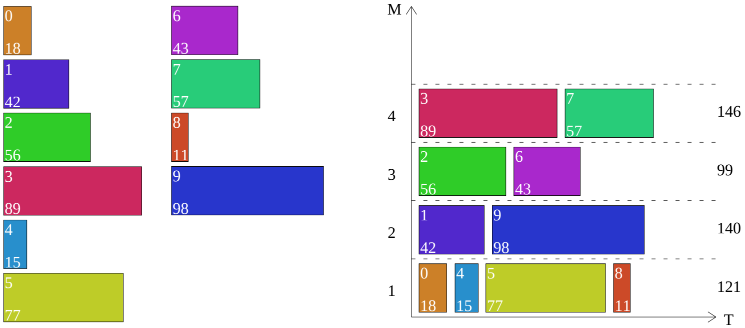
Exercise: improve this solution using previously mentioned ideas



Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

Question 3: what to do when x is locally optimal with N_2?

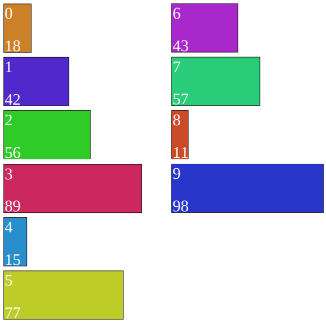
Exercise: improve this solution using previously mentioned ideas

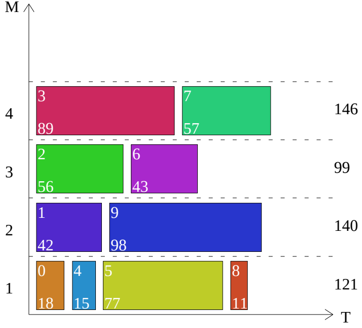


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

Question 3: what to do when x is locally optimal with N_2?

Exercise: improve this solution using previously mentioned ideas

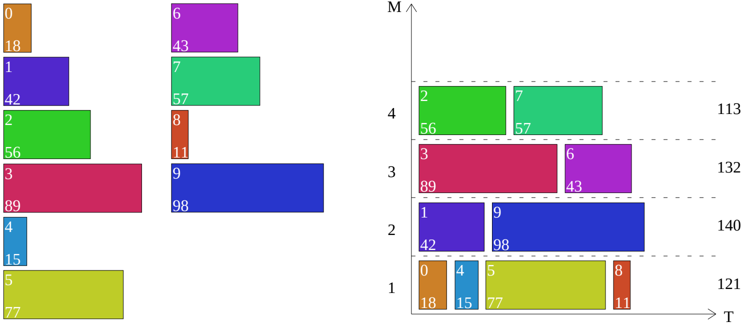




Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

Question 3: what to do when x is locally optimal with N_2?

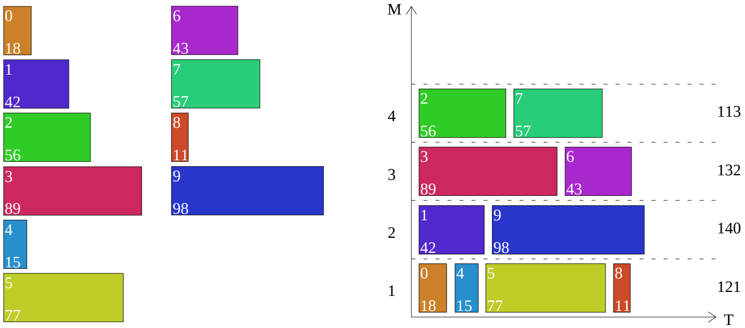
Exercise: improve this solution using previously mentioned ideas



Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

Question 3: what to do when x is locally optimal with N_2?

Exercise: improve this solution using previously mentioned ideas

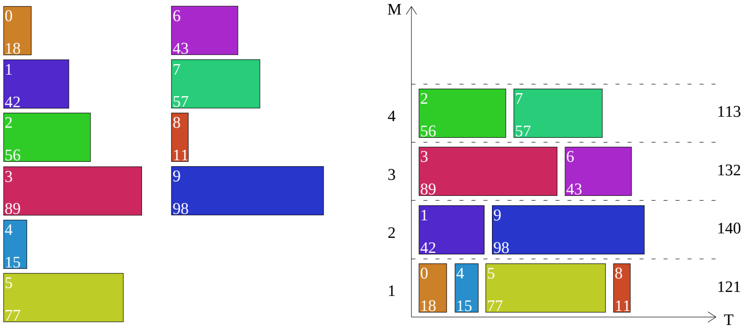


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

Then we are stuck with both neighbourhoods!

Question 3: what to do when x is locally optimal with N_2?

Exercise: improve this solution using previously mentioned ideas



Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

Then we are stuck with both neighbourhoods!
This solution could still be improved though ($z(x^*) = 131$).

Question 3: what to do when x is locally optimal with N_2?

- Use N_3!

Question 3: what to do when x is locally optimal with N_2?

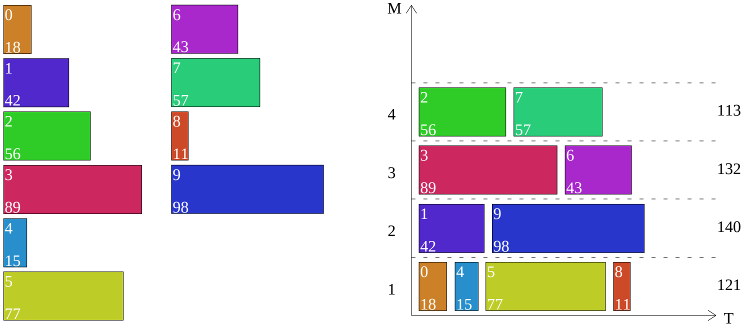
- Use N_3!
- Concerns regarding N_2 and N_3:
  - Similar to those regarding N_1 and N_2
  - N_3 should aim at escaping from local optima for N_2

Question 3: what to do when x is locally optimal with N_2?

- Use N_3!
- Concerns regarding N_2 and N_3:
  - Similar to those regarding N_1 and N_2
  - N_3 should aim at escaping from local optima for N_2
- Concerns regarding N_1 and N_3:
  - Similar to those regarding N_1 and N_2 (!)
  - If we have to use N_3, this is because N_2 could not escape from a local optimum for N_1
  - N_3 should aim at escaping from this local optimum for N_2 and N_1
- And so on

Question 3: what to do when x is locally optimal with N_2?

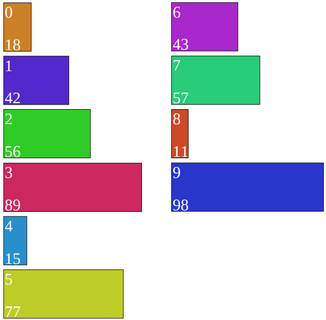
Exercise: find a useful N_3

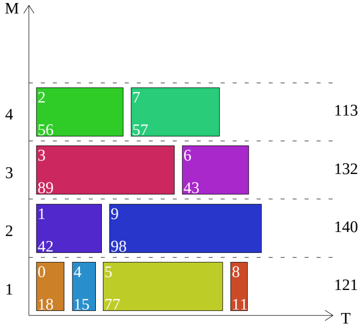


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

Question 3: what to do when x is locally optimal with N_2?

Exercise: find a useful N_3





Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

One possibility is to exchange job 1 against jobs 0 and 4.

Exercise: write $bestSwap1Vs2(x)$

(Returns the best neighbour which can be obtained by exchanging 1 job against 2)

Exercise: write *bestSwap*1*Vs*2(x)

(Returns the best neighbour which can be obtained by exchanging 1 job against 2)

bestSwap1Vs2(x)

```
bestDuration ← ∞
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        if x[i] ≠ x[j] then
            for k ← 0 to n − 1 do
                if x[j] = x[k] then
                    x[i] ↔ x[j]
                    x[k] ← x[j]
                    if getDuration(x) < bestDuration then
                        bestDuration ← getDuration(x)
                        bestJob1 ← i
                        bestJob2 ← j
                        bestJob3 ← k
                    end if
                    x[i] ↔ x[j]
                    x[k] ← x[j]
                end if
            end for
        end if
    end for
end for
x′ ← x
x′[bestJob1] ↔ x′[bestJob2]
x′[bestJob3] ← x′[bestJob2]
Return x′
```

Question 4: What to do when N_{k+1} allowed to escape?

Exercise: what can we do when N_{k+1} allowed to escape from a local optimum for N_k?

Question 4: What to do when N_{k+1} allowed to escape?

Exercise: what can we do when N_{k+1} allowed to escape from a local optimum for N_k?

(at least) 3 possibilities:

- Continue with N_{k+1} until we meet a local optimum
- Continue with N_k, using N_{k+1} when necessary
- Fall back to N_1

Question 4: What to do when N_{k+1} allowed to escape?

Exercise: what can we do when N_{k+1} allowed to escape from a local optimum for N_k?

(at least) 3 possibilities:

- Continue with N_{k+1} until we meet a local optimum
- Continue with N_k, using N_{k+1} when necessary
- Fall back to N_1

- How it is done in VND: fall back to N_1
- Motivation: spend as little time as we can in expensive neighbourhoods

It has to end somewhere

Assume that we already have:

- κ neighbourhoods named $N_1, N_2, ..., N_\kappa$
- Associated steepest descents: $N_k^*(x)$ returns the best neighbour of solution x in neighbourhood N_k
- A construction heuristic *buildSolution*()

Exercise: write the abstract algorithm for VND

VND abstract algorithm

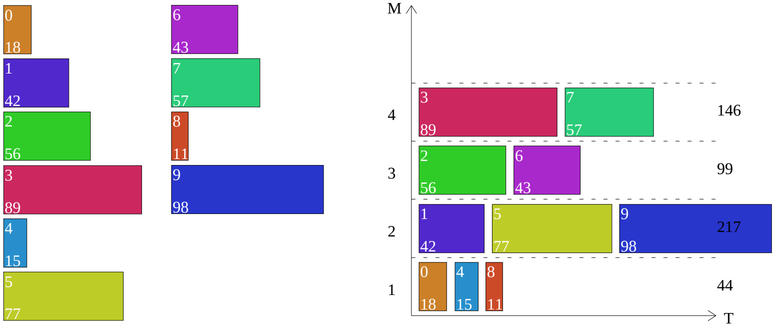
VND()

$x \leftarrow buildSolution()$
$k \leftarrow 1$
while $k \leq \kappa$ **do**
    $x' \leftarrow N_k^*(x)$
    if $z(x') < z(x)$ **then**
        $x \leftarrow x'$
        $k \leftarrow 1$
    else
        $k \leftarrow k + 1$
    end if
end while
Return x

Note that when the algorithm is over, x is a local optimum for all κ neighbourhoods

Full VND

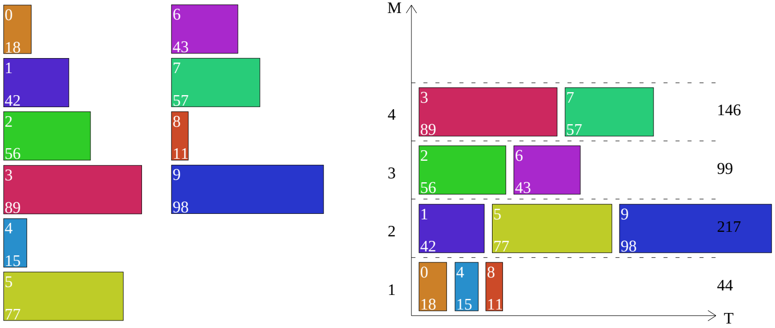
Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods



Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	2	3	4	1	2

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

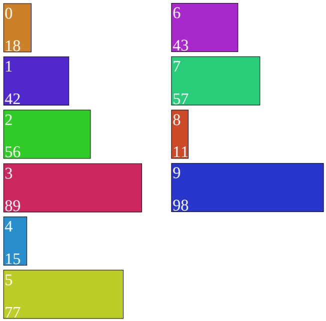


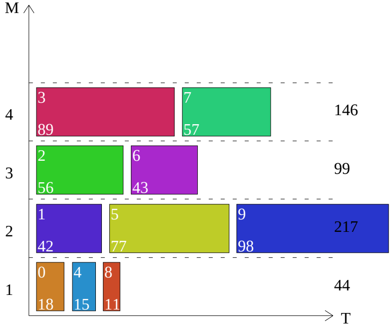
Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	2	3	4	1	2

$k = 1$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods



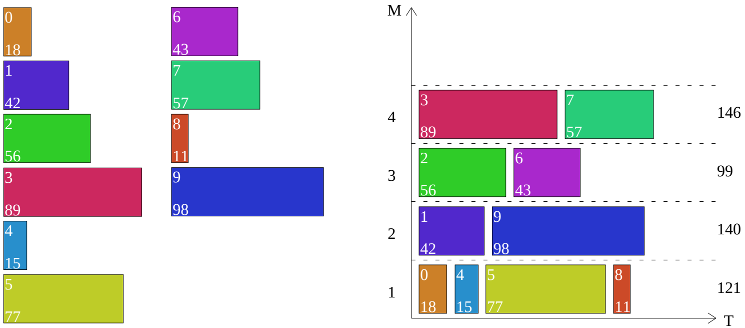


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	2	3	4	1	2

$k = 1$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

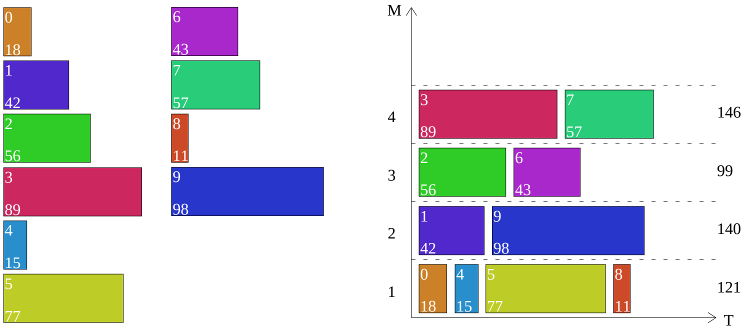


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

$k = 1$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

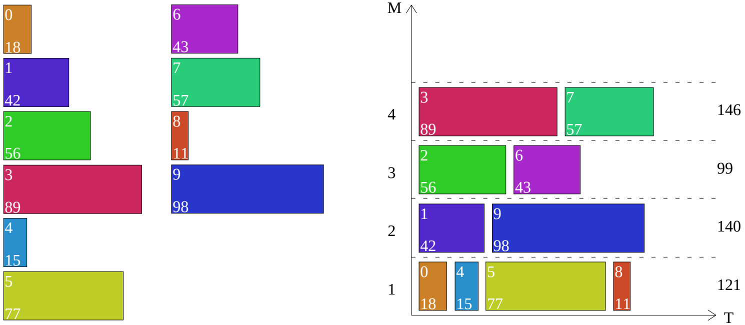


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

$k = 1$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

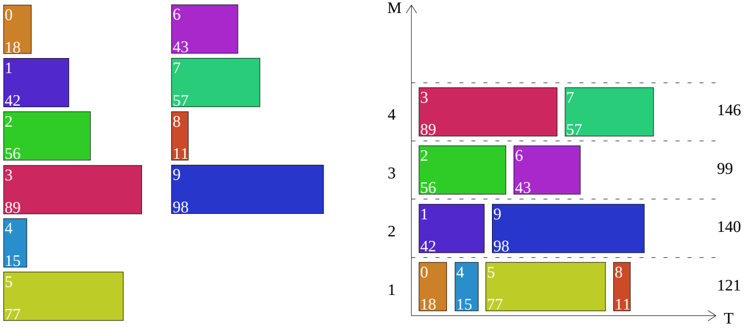


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

$k = 2$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

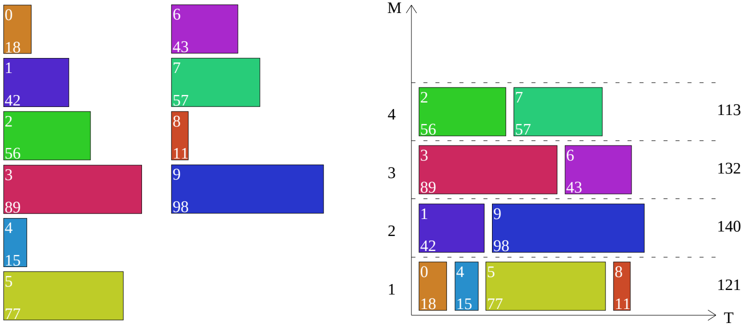


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	3	4	1	1	3	4	1	2

$k = 2$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

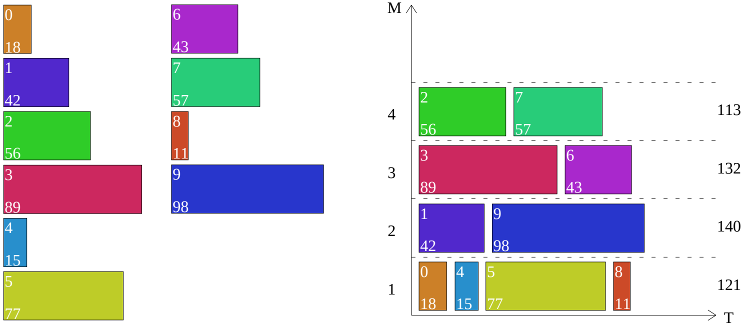


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

$k = 2$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

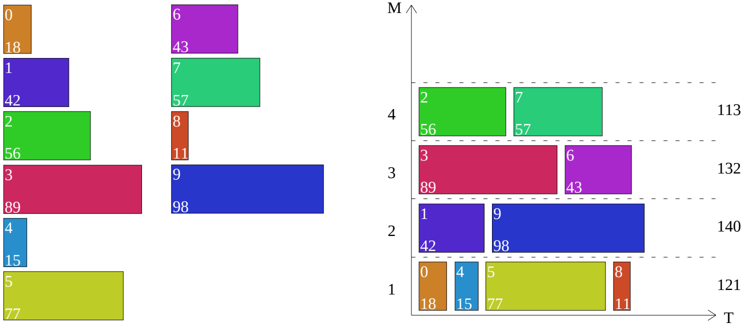


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

$k = 1$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

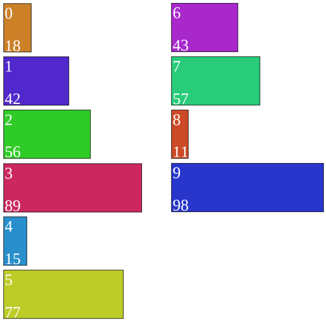


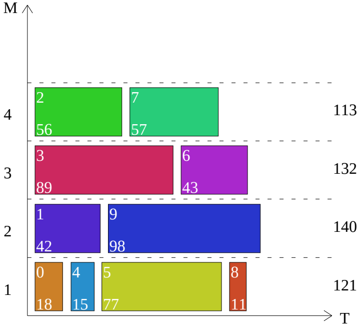
Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

$k = 2$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods



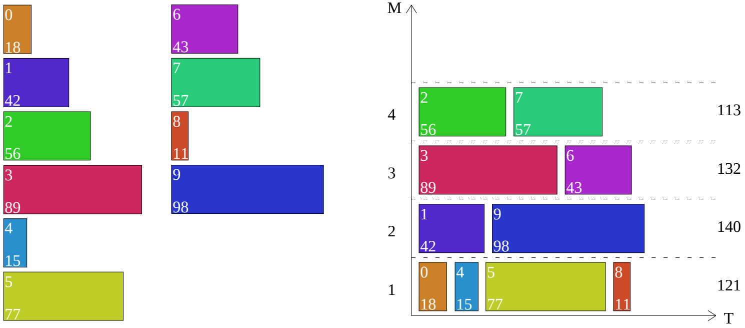


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

$k = 3$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

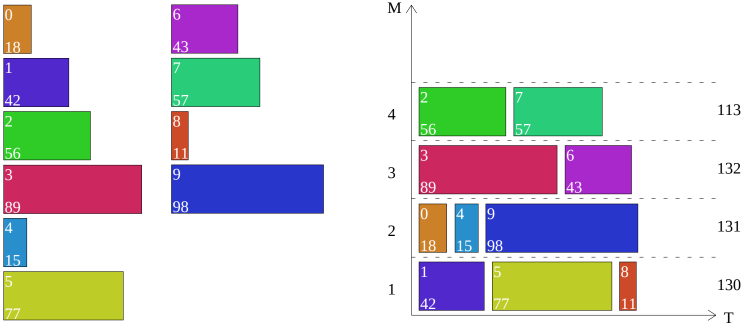


Job	0	1	2	3	4	5	6	7	8	9
Machine	1	2	4	3	1	1	3	4	1	2

$k = 3$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

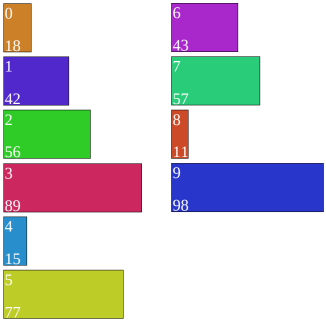


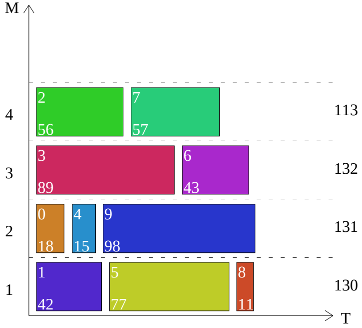
Job	0	1	2	3	4	5	6	7	8	9
Machine	2	1	4	3	2	1	3	4	1	2

$k = 3$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods



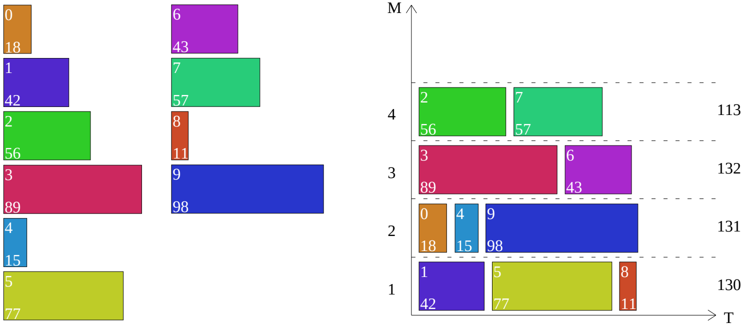


Job	0	1	2	3	4	5	6	7	8	9
Machine	2	1	4	3	2	1	3	4	1	2

$k = 1$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

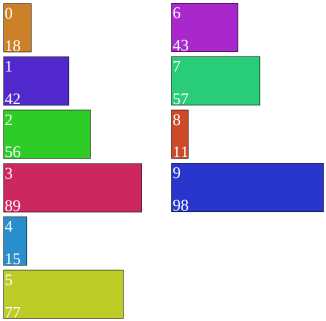


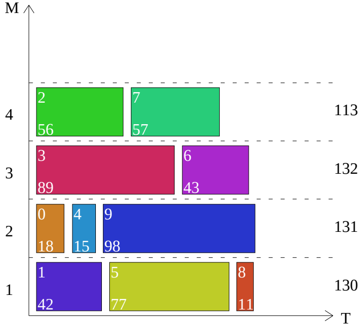
Job	0	1	2	3	4	5	6	7	8	9
Machine	2	1	4	3	2	1	3	4	1	2

$k = 2$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods



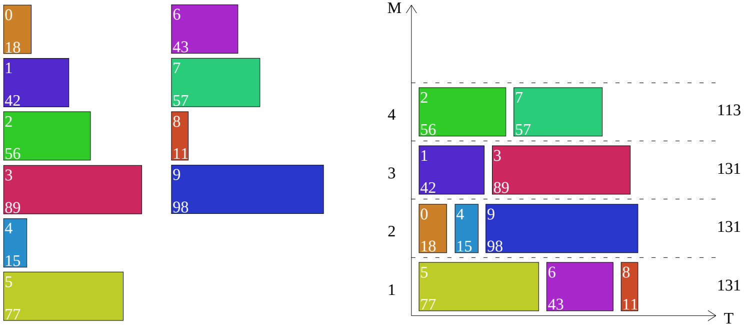


Job	0	1	2	3	4	5	6	7	8	9
Machine	2	1	4	3	2	1	3	4	1	2

$k = 2$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

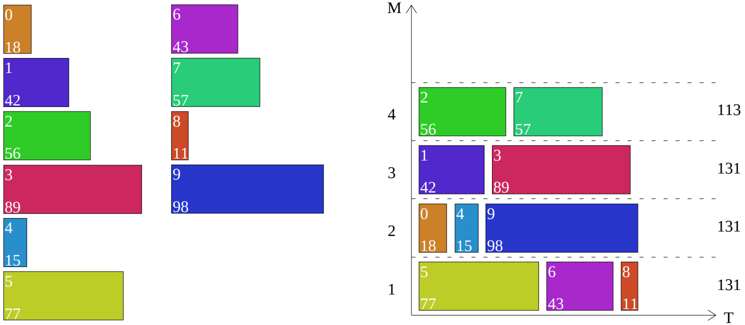


Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	4	3	2	1	1	4	1	2

$k = 2$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods

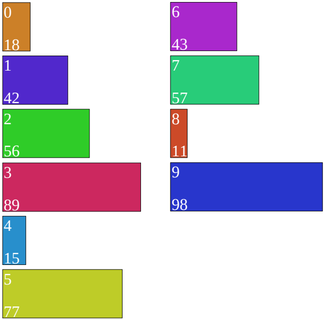


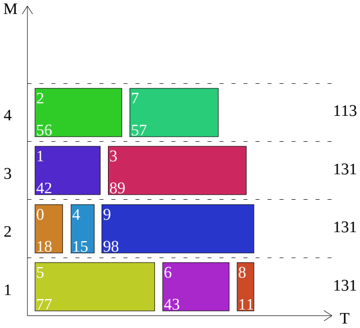
Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	4	3	2	1	1	4	1	2

$k = 1$

Full VND

Exercise: perform a Variable Neighbourhood Descent with the 3 previously introduced neighbourhoods





Job	0	1	2	3	4	5	6	7	8	9
Machine	2	3	4	3	2	1	1	4	1	2

$k = 1$

Optimum found!

Quantitative analysis: comparison with Tabu Search

$N_4 = N_1 \times N_1$

Instance (m, n)	TLS $= 6$	$\kappa = 4$	$\kappa = 3$	Steepest Descent
4, 8	109	109	109	114
4, 10	131	131	131	146
4, 12	219	217	217	253
5, 8	74	74	74	74
5, 10	117	122	122	122
5, 12	135	135	135	157
10, 20	125	135	135	135
10, 30	188	193	203	203
10, 40	212	214	268	268
10, 50	282	295	295	295
10, 60	361	366	393	393

Quantitative analysis: comparison with Tabu Search

$N_4 = N_1 \times N_1$

Instance (m, n)	TLS $= 6$	$\kappa = 4$	$\kappa = 3$	Steepest Descent
4, 8	109	109	109	114
4, 10	131	131	131	146
4, 12	219	217	217	253
5, 8	74	74	74	74
5, 10	117	122	122	122
5, 12	135	135	135	157
10, 20	125	135	135	135
10, 30	188	193	203	203
10, 40	212	214	268	268
10, 50	282	295	295	295
10, 60	361	366	393	393

- Without surprise, results are better with a fourth, more powerful neighbourhood
- This VND seems to work better on smaller instances
- N_4 is starting to be too slow on the bigger instances (> 10 s)
- With $\kappa = 3$ it is very quick but can't escape local optima well
- But maybe with a different starting solution it would be better. . .

Variable Neighbourhood Descent: Discussion

Qualitative analysis check list

- Clarity, Modularity, Simplicity
- Intensification
- Diversification

Your opinion

- Questions
- Suggestions

Outline

Iterated Local Search: motivation

Remember the 3 suggestions!

- Clarity, modularity, simplicity
- Intensification
- Diversification

Iterated Local Search: motivation

Remember the 3 suggestions!

- Clarity, modularity, simplicity
- Intensification
- Diversification

Methods detailed until now might be seen as:

- Empirically built around unsatisfying standalone local searches
- A bit weak on diversification (although this can be compensated, e.g. long-term memory for TS)

Iterated Local Search: principles

- Iterated Local Search (ILS) is a very generic metaheuristic
- It aims at efficiently providing intensification and diversification, in a simple and modular way
- It consists in improving local search "by iteration"

Iterated Local Search: principles

- Iterated Local Search (ILS) is a very generic metaheuristic
- It aims at efficiently providing intensification and diversification, in a simple and modular way
- It consists in improving local search "by iteration"

ILS general scheme: one iteration

1. $x' \leftarrow \textit{diversify}(x)$
2. $x'' \leftarrow \textit{intensify}(x')$
3. $x \leftarrow \textit{acceptanceDecision}(x, x'')$

The whole search simply consists in performing a certain amount of iterations.

(N.B.: in ILS diversification is achieved through solution perturbation)

Iterated Local Search: intensification

Use here your favorite local search.
Be careful though, time is incompressible!

Suppose we have 1 minute of CPU time...

- If we have a very good local search that takes 5 seconds:
  - It will probably find reasonably good solutions
  - But we can only perform it for 12 iterations
  - So even a very good diversification process might be insufficient to find a very good solution

Iterated Local Search: intensification

Use here your favorite local search.
Be careful though, time is incompressible!

Suppose we have 1 minute of CPU time...

- If we have a very good local search that takes 5 seconds:
  - It will probably find reasonably good solutions
  - But we can only perform it for 12 iterations
  - So even a very good diversification process might be insufficient to find a very good solution
- If we have a very quick local search that takes 0.01 second:
  - It is probably also very bad
  - It can find very good solutions if we are lucky and have a very good diversification process
  - We can perform 6000 iterations, with the risk of finding 6000 bad solutions

Iterated Local Search: intensification

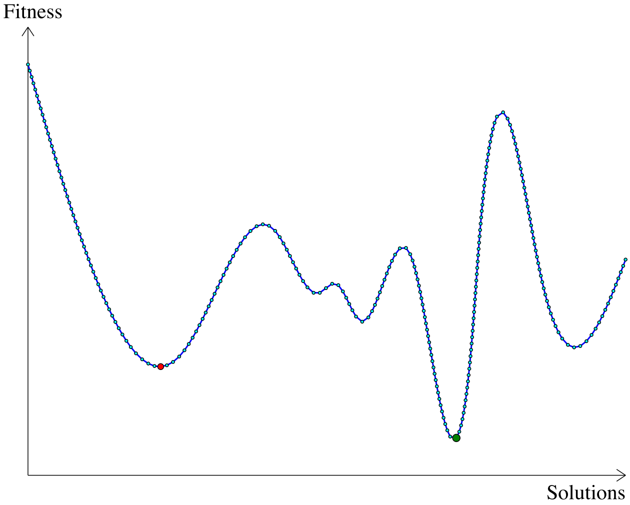
Use here your favorite local search.
Be careful though, time is incompressible!

Suppose we have 1 minute of CPU time...

- If we have a very good local search that takes 5 seconds:
  - It will probably find reasonably good solutions
  - But we can only perform it for 12 iterations
  - So even a very good diversification process might be insufficient to find a very good solution
- If we have a very quick local search that takes 0.01 second:
  - It is probably also very bad
  - It can find very good solutions if we are lucky and have a very good diversification process
  - We can perform 6000 iterations, with the risk of finding 6000 bad solutions

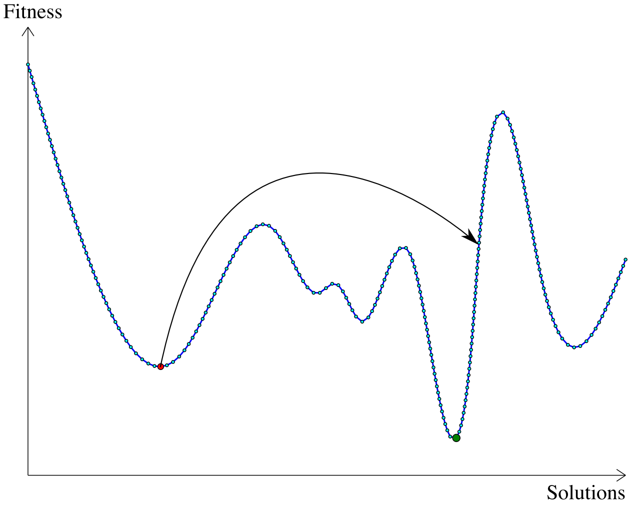
Finding a good compromise is not always easy!

Iterated Local Search: diversification



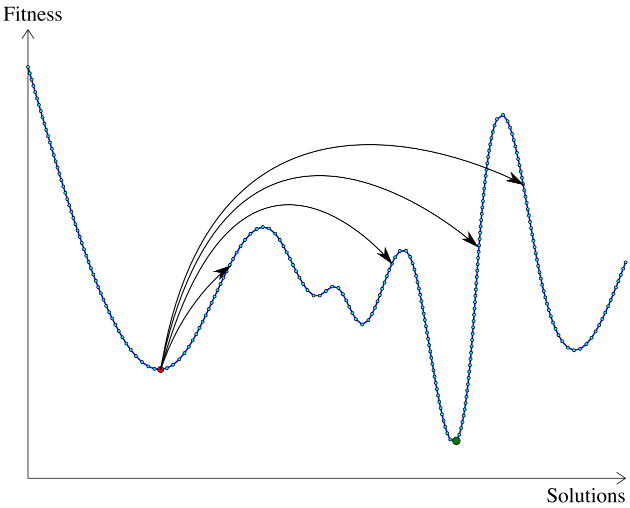
Special care should be taken to diversify properly.

Iterated Local Search: diversification



Special care should be taken to diversify properly.

Iterated Local Search: diversification



Special care should be taken to diversify properly.

Iterated Local Search: diversification

How perturbation is achieved

- Perturbation usually consists in random move(s) in "higher order" neighbourhoods
- If perturbation is too weak, local optimum cannot be escaped
- If perturbation is too strong, ILS is equivalent to random restarts

Iterated Local Search: diversification

How perturbation is achieved

- Perturbation usually consists in random move(s) in "higher order" neighbourhoods
- If perturbation is too weak, local optimum cannot be escaped
- If perturbation is too strong, ILS is equivalent to random restarts

Exercise: propose an ILS scheme for the $P||C_{max}$

Iterated Local Search: diversification

How perturbation is achieved

- Perturbation usually consists in random move(s) in "higher order" neighbourhoods
- If perturbation is too weak, local optimum cannot be escaped
- If perturbation is too strong, ILS is equivalent to random restarts

Exercise: propose an ILS scheme for the $P||C_{max}$

One possible way to do it

1. Perturbation:
2. Local search:
3. Acceptance:

Iterated Local Search: diversification

How perturbation is achieved

- Perturbation usually consists in random move(s) in "higher order" neighbourhoods
- If perturbation is too weak, local optimum cannot be escaped
- If perturbation is too strong, ILS is equivalent to random restarts

Exercise: propose an ILS scheme for the $P||C_{max}$

One possible way to do it

1. Perturbation: perform m random moves on the "job move" neighbourhood
2. Local search: steepest descent on the "job move" neighbourhood
3. Acceptance: accept only improvements

Iterated Local Search: diversification issues

Another issue: the right perturbation at the right time

- The need for diversification might be context-dependent, e.g.
  - In a non-promising region, perturb more
  - In a promising region, perturb less

Iterated Local Search: diversification issues

Another issue: the right perturbation at the right time

- The need for diversification might be context-dependent, e.g.
  - In a non-promising region, perturb more
  - In a promising region, perturb less
- Dynamically tuning perturbation power can be useful
- Search history can be involved in the perturbation process

Iterated Local Search: abstract algorithm

ILS()

$x \leftarrow \mathit{constructionHeuristic}()$
$x \leftarrow \mathit{localSearch}(x)$
$x^* \leftarrow x$
while stopping criterion not met **do**
  $x' \leftarrow \mathit{perturbation}(x, \mathit{history})$
  $x'' \leftarrow \mathit{localSearch}(x')$
  $x \leftarrow \mathit{acceptanceDecision}(x, x'', \mathit{history})$
  if $z(x) < z(x^*)$ **then**
    $x^* \leftarrow x$
  end if
end while
Return x^*

Outline

Introducing Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is a popular metaheuristic which can be seen as a specific ILS.

VNS specials

- In VNS, the perturbation steps are called <u>Shaking</u>
- The neighbourhood used for shaking changes during the search, hence the name
- The neighbourhoods used for shaking are selected in a similar way as with VND

Introducing Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is a popular metaheuristic which can be seen as a specific ILS.

VNS specials

- In VNS, the perturbation steps are called <u>Shaking</u>
- The neighbourhood used for shaking changes during the search, hence the name
- The neighbourhoods used for shaking are selected in a similar way as with VND

More precisely, we have:

- κ <u>shaking</u> neighbourhoods $N_1, \ldots, N_\kappa$
- Associated random procedures: $N_k^r(x)$ returns a random element from $N_k(x)$

Choosing neighbourhoods

- As with VND, one aim is to favor the use of small neighbourhoods
- Larger ones are used only when smaller ones are ineffective
- It is common to use so-called nested neighbourhoods, i.e. $\forall k, x : N_k(x) \subset N_{k+1}(x)$
- This way small perturbations are favored

Choosing neighbourhoods

- As with VND, one aim is to favor the use of small neighbourhoods
- Larger ones are used only when smaller ones are ineffective
- It is common to use so-called nested neighbourhoods, i.e. $\forall k, x : N_k(x) \subset N_{k+1}(x)$
- This way small perturbations are favored

Consider the following neighbourhoods for the $P||C_{max}$:

- N_1: move a job to another machine
- N_k: perform k times a N_1 move

Choosing neighbourhoods

Exercise: are the following statements true?

- $N_1 \subset N_2$

Choosing neighbourhoods

Exercise: are the following statements true?

- $N_1 \subset N_2$
  Yes: move the same job from a to b, then from b to c

Choosing neighbourhoods

Exercise: are the following statements true?

- $N_1 \subset N_2$
  Yes: move the same job from a to b, then from b to c
- $N_1 \subset N_3$

Choosing neighbourhoods

Exercise: are the following statements true?

- $N_1 \subset N_2$
  Yes: move the same job from a to b, then from b to c
- $N_1 \subset N_3$
  Yes: first perform a move, then the reverse move, then perform a move

Choosing neighbourhoods

Exercise: are the following statements true?

- $N_1 \subset N_2$
  Yes: move the same job from a to b, then from b to c
- $N_1 \subset N_3$
  Yes: first perform a move, then the reverse move, then perform a move

However:

- Those are just special cases, with very low probability
- This might not work with another problem

Choosing neighbourhoods

Exercise: are the following statements true?

- $N_1 \subset N_2$
  Yes: move the same job from a to b, then from b to c
- $N_1 \subset N_3$
  Yes: first perform a move, then the reverse move, then perform a move

However:

- Those are just special cases, with very low probability
- This might not work with another problem

There is a common way to ensure that "small" moves will be performed often, even with "large" neighbourhoods:

- N_1: perform an elementary move (problem-independent)
- N_k: perform between 1 and k elementary moves

Generating a random neighbour for $P||C_{max}$

random() returns a random number in $[0, 1)$

Exercise: Generate a random integer in $[a, b]$

$randomInteger(a, b)$

Generating a random neighbour for $P||C_{max}$

random() returns a random number in $[0, 1)$

Exercise: Generate a random integer in $[a, b]$

***randomInteger*(a, b)**

1: **return** $a + \lfloor random() \cdot (b - a + 1) \rfloor$

Exercise: 1 to k times, randomly move a job

$N_k^r(x)$

Generating a random neighbour for $P||C_{max}$

random() returns a random number in $[0, 1)$

Exercise: Generate a random integer in $[a, b]$

$randomInteger(a, b)$

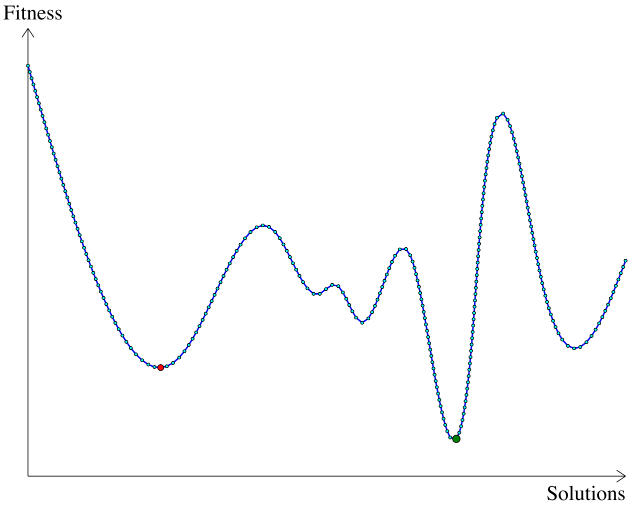
return $a + \lfloor random() \cdot (b - a + 1) \rfloor$

Exercise: 1 to k times, randomly move a job

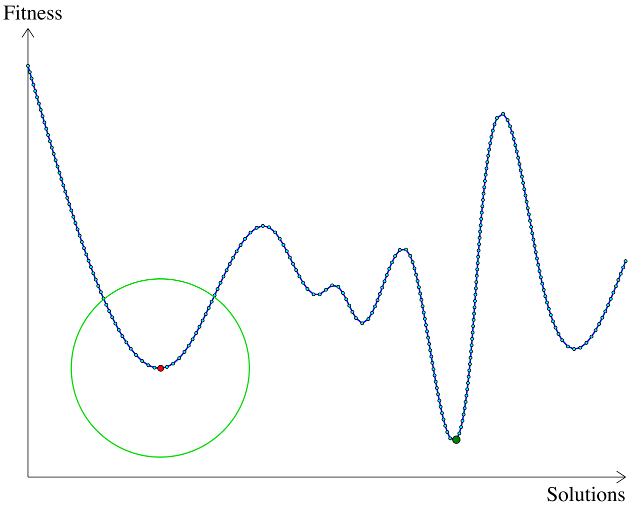
$N_k^r(x)$

$x' \leftarrow x$
$q \leftarrow randomInteger(1, k)$
for $c \leftarrow 1$ to q **do**
    $i \leftarrow randomInteger(0, n-1)$
    $j \leftarrow randomInteger(1, m)$
    $x'[i] \leftarrow j$
end for
return x'

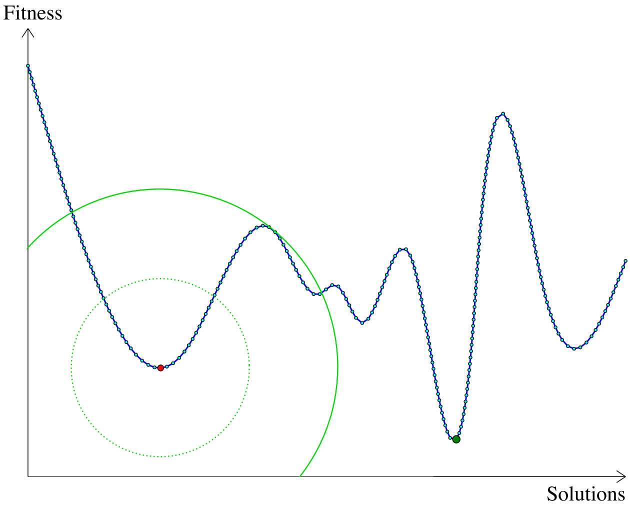
Nested neighbourhoods



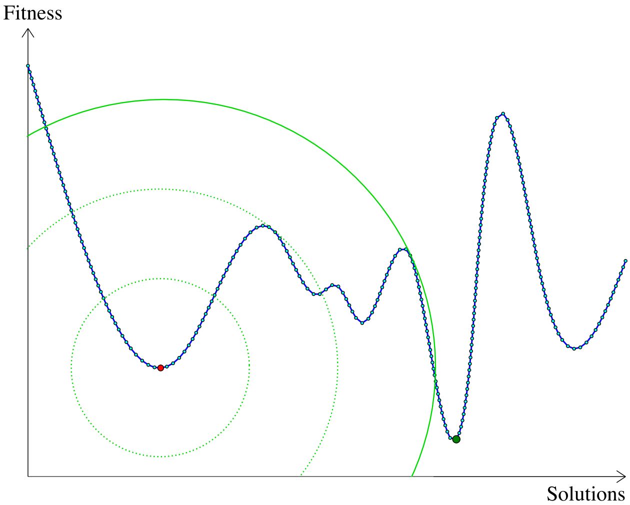
Nested neighbourhoods



Nested neighbourhoods



Nested neighbourhoods



Variable Neighbourhood Search: abstract algorithm

VNS()

$x \leftarrow$ *buildSolution*()
$x^* \leftarrow x$
$k \leftarrow 1$
while stopping criterion not met **do**
    $x' \leftarrow N_k^r(x)$
    $x'' \leftarrow$ *localSearch*(x')
    if *acceptanceDecision*(x, x'') **then**
        $x \leftarrow x''$
        $k \leftarrow 1$
        if $z(x) < z(x^*)$ **then**
            $x^* \leftarrow x$
        end if
    else
        $k \leftarrow 1 + (k \bmod \kappa)$
    end if
end while
Return x^*

Things to know about ILS/VNS

Typical acceptance criteria

- Always accept improvements
- Threshold acceptance: accept slight degradations in certain cases, e.g. after a certain number of iterations without any improvement
- Simulated Annealing

Things to know about ILS/VNS

Typical acceptance criteria

- Always accept improvements
- Threshold acceptance: accept slight degradations in certain cases, e.g. after a certain number of iterations without any improvement
- Simulated Annealing

Common stopping criteria

- Certain number of iterations
- Certain number of iterations without any improvement
- CPU time limit

Quantitative analysis: comparison with Tabu Search

1 run, 500 iterations, local search = steepest descent on N_1

Instance (m, n)	TLS = 6	ILS (m moves)	VNS ($\kappa = m$)	Steepest Descent
4, 8	109	109	109	114
4, 10	131	131	131	146
4, 12	219	217	217	253
5, 8	74	74	74	74
5, 10	117	117	117	122
5, 12	135	135	135	157
10, 20	125	125	125	135
10, 30	188	186	185	203
10, 40	212	212	212	268
10, 50	282	283	282	295
10, 60	361	361	360	393

ILS and VNS: Discussion

Qualitative analysis check list

- Clarity, Modularity, Simplicity
- Intensification
- Diversification

Your opinion

- Questions
- Suggestions