Rapport de Projet ECMA

Romain Barrault et Cyril Nederveen

February 5, 2023

1 Heuristique de résolution

La conception de l'heuristique de résolution se décompose en deux parties : la conception d'une heuristique permettant d'aboutir à une solution réalisable, suivi d'une heuristique d'amélioration à l'aide d'une recherche locale.

1.1 Obtention d'une solution initiale

Formellement, la meilleure manière d'aboutir à une solution réalisable initiale serait de passer par de la programmation par contraintes. Nous avons choisi de concevoir nous-mêmes une heuristique qui ne garantit pas l'obtention d'une solution réalisable, mais que nous pouvions espérer assez pertinente pour fonctionner sur les instances du problème ; ceci ayant été le cas, nous nous sommes satisfaits de cette dernière.

Cet algorithme, comme une partie de celui de recherche locale détaillé plus bas, repose sur le fait que les sous-problèmes liés à la robustesse des contraintes de poids et des longueurs entre les sommets sont des problèmes analogues à la relaxation continue du problème du sac à dos, pour lesquels les solutions optimales sont relativement faciles à déterminer. Ainsi, les "pires scénarios" d'instanciation de poids et de longueur se déterminent facilement :

- Pour les incertitudes sur les distances, on crée un dictionnaire dont les clefs sont les couples $(i, j) \in [\![1; n]\!]$ et les valeurs sont égales à $\hat{l_i} + \hat{l_j}$, puis on le trie par ordre décroissant de valeur. Par la suite, on instancie dans l'ordre ainsi défini et pour tout couple (i, j) tel que $x_{ij} = 1$ tous les δ^1_{ij} égaux à 3 (leur valeur maximale) tant que c'est possible ; quand cela ne l'est plus, on instancie le δ^1_{ij} courant à $L - \sum_{ij \in E} \delta^1_{ij} < 3$, et on attribue une valeur nulle à tous les autres δ^1_{ij}.
- De même pour les incertitudes sur les poids, à l'exception que les valeurs à partir desquels trier sont les w_v, que la valeur maximale de δ^2_v n'est plus 3 mais W_v et que leur somme n'est plus égale à L mais W.

Par ailleurs, la vérification de la robustesse sur les poids fera partie du critère d'admissibilité d'une solution. Celle sur les distances sera nécessaire pour déterminer le coût réel d'une solution.

Le principe est le suivant : on construit les ensembles partitionnant l'ensemble des points de manière itérative. Une fois qu'un ensemble de la partition est initialisé à l'ensemble vide, on crée une variable nulle $worst\_w$ qui symbolisera la somme des poids des sommets de l'ensemble dans le pire des scénarios, une autre variable nulle $robust\_w$ qui symbolisera la somme des coefficients δ^2_v (à maintenir inférieur à W), puis on réalise les étapes suivantes :

- On vérifie qu'il reste des sommets à placer dans un ensemble de la partition ; ils doivent vérifier $w_v(1 + \min(W_v, W)) \leq B$.
- On commence par placer dans la partition celui dont le poids w_v est maximal ; la variable δ^2_v est symboliquement instanciée égale à $\min(W_v, W)$ en incrémentant d'autant la variable $robust\_w$ et en incrémentant $worst\_w$ de $w_v(1 + \min(W_v, W))$.

- On réitère à partir de l'ensemble des sommets vérifiant :
  $w_v(1 + min(W_v, W - robust\_w)) \leq B - worst\_w$, on sélectionne parmi les sommets vérifiant cette assertion celui de poids maximal, et on retourne à l'étape 2.

- Une fois qu'on ne peut plus ajouter aucun sommet à l'ensemble courant, il est considéré comme achevé et on passe à l'ensemble suivant. On recommence alors à partir de l'étape 1 ; on répète ces étapes tant qu'il y a des sommets à placer ou tant que l'on n'a pas créé et rempli K ensembles.

L'heuristique échoue à aboutir à une solution réalisable dans ce dernier cas, mais il n'a pas été observé sur l'ensemble des 55 instances à notre disposition. Elle repose le principe suivant : en commençant à attribuer à un ensemble les sommets par ordre décroissant de poids, il y a plus de chances de combler au mieux les ensembles avec les sommets de poids plus petits.

Nous sommes alors en possession d'une solution réalisable au regard de l'incertitude sur les poids. Il reste à déterminer sa valeur réelle en instanciant les variables δ_{ij}^1 conformément à la solution de la relaxation continue du problème du sac à dos (expliquée en début de sous-section).

Nous avons alors à notre disposition une solution réalisable au regard du problème d'optimisation robuste, et nous pouvons l'améliorer à l'aide d'une heuristique de recherche locale décrite ci-après.

1.2 Amélioration à l'aide d'une métaheuristique

Le voisinage d'une solution considéré sera défini à l'aide de deux opérateurs : *shift* (modélisé par la fonction *changed_set*) et *swap* (modélisé par la fonction *swapped*). Le premier considère un voisin obtenu en changeant l'ensemble auquel un sommet appartient, le second considère un voisin obtenu en échangeant les deux ensembles d'appartenance de deux sommets disjoints. A toute génération de voisin, la vérification de la robustesse sur les poids sera effectuée, de manière à ce qu'un voisin ne soit considéré comme réalisable que s'il l'est toujours dans les "pires scénarios" pesant sur les poids des sommets contenus dans les ensembles constituant sa partition.

L'amélioration de la solution se fera à l'aide d'une recherche à voisinage variable, ou **VNS**. Le principe est le suivant :

- On part de la solution établie précédemment ; elle est conforme à la robustesse pesant sur les contraintes de poids et sur la longueur des arcs.

- On génère un voisin aléatoire (réalisable) dans le voisinage de la solution courante ; on considère alors son propre voisinage, et on en garantit l'exploration complète afin d'aboutir à un optimum local. Attention ici, pour juger de l'optimalité de cette solution, on n'a pas touché aux valeurs de δ_{ij}^1 donc cet optimum n'est pas nécessairement pertinent au regard des incertitudes sur les longueurs ;

- On répète cette étape de manière à visiter au moins 5 voisinages, ou à visiter autant de voisinages possibles pendant au moins 5 secondes. L'optimum local à laquelle la VNS aboutit est alors celui de coût moindre parmi les optima locaux déterminés.

- On instancie les δ_{ij}^1 correspondant au pire scénario possible pour cet optimum local et on met à jour son coût réel. Si son coût est inférieur à celui de la solution courante alors cette solution devient la solution courante.

- On retourne à l'étape 2.

On réalise alors ces différentes étapes pendant une durée prédéterminée, à l'issue de laquelle nous obtenons une solution améliorée dans la plupart des cas. Les résultats obtenus sont consignés dans la section Résultats.
Par défaut, les paramètres de calcul sont faits pour permettre jusqu'à 10 améliorations de la solution courante, et une visite d'au moins 5 voisinages complets lors de la VNS.

Implémentation et amélioration des modèles

Nous présentons dans cette section les moyens mis en oeuvre pour l'implémentation des méthodes déterminées lors de la modélisation papier. Lors de la phase d'implémentation, nous avons réalisé quelques amélioration des méthodes dans le but d'accélérer la résolution.

Premiers résultats de résolution

Pour chaque implémentation, nous nous sommes servi de l'instance `10_ulysses_3.tsp` (pour laquelle nous avions à disposition la valeur de l'objectif et la solution associée) pour tester et valider le programme. Lorsque il s'avérait difficile de comprendre l'origine d'une erreur, nous menions des tests sur une autre instance plus simple que nous avons créé :

```
n = 3
L = 2
W = 9
K = 2
B = 21
w_v = [4, 14, 3]
W_v = [0, 0, 4]
lh = [10, 7, 9]
coordinates = [
38.24 20.42 ;
39.57 26.15 ;
40.56 25.32 ]
```

Celle-ci donne des solutions triviales qui sont les clusters $\{1\}$ et $\{2,3\}$ pour le problème statique, et les clusters $\{1,2\}$ et $\{3\}$ pour le problème robuste.

De manière générale, nous avons ajouté les contraintes suivantes afin de réduire la symétrie par rapport au variables x_{ij} (qui correspond également à x_{ji}).

```
for j in 1:n
    for i in j+1:n
        x[i, j] = x[j, i]
    end
end
```

Nous avons également supprimé les contraintes suivantes car dans le cas de la minimisation, celles-ci sont toujours vérifiées.

```
@constraint(m, [i in 1:n, j in i+1:n, k in 1:K], y[k,i] - y[k,j] <= 1-x[i,j])
@constraint(m, [i in 1:n, j in i+1:n, k in 1:K], -y[k,i] + y[k,j] <= 1-x[i,j])
```

Dans chacune des méthodes, nous faisions face à un problème de symétrie par rapport aux clusters. En effet, pour une solution donnée, toute permutation des clusters correspond à la même solution. D'un point de vue algorithmique cela pose problème car beaucoup de cas redondant sont traités. Pour éviter cela, nous avons tenter d'imposer un ordre d'affectation des noeuds aux clusters de sorte que le noeud n°1 soit forcément dans le premier cluster, ou de manière équivalente $y_k 1 = 0$ pour $k > 1$.

Plus généralement, on impose que $y_k i = 0$ pour $k > i$. Cela revient donc à supprimer les variables correspondantes dans le programme comme suit :

```
@variable(m, y[k in 1:K, i in k:n], Bin)
```

au lieu de :

```
@variable(m, y[k in 1:K, i in 1:n], Bin)
```

Au total, on enlève $K(K-1)/2$ variables de type y_{ki}. Malheureusement il semble que d'un point de vue pratique, K ne soit pas assez grand pour que cela ait un effet significatif sur le temps de résolution.

Branch & cut et Plans coupants

Les méthodes branch & cut et plans coupants diffèrent par le fait que la résolution du problème maître reprend du début dans plans coupants. Pour faire apparaître cette distinction dans l'implémentation, nous avons opté pour des callbacks dans le cas du branch & cut, et une boucle while dans le cas de plans coupants.

Dans les deux cas, on fait appel aux fonctions `fast_sp1_solve` et `fast_sp2k_solve` qui permettent respectivement de résoudre les sous-problèmes $SP1$ et $SP2_k$. Ainsi :

- `fast_sp1_solve` : prend en argument l'instance courante, la solution x courante, les données l_{ij} (pour éviter de recalculer à chaque fois) et une limite de temps pour la résolution. Renvoie la valeur et la solution associées à $(SP1)$

- `fast_sp2k_solve` : prend en argument l'instance courante, la solution y courante et une limite de temps pour la résolution. Renvoie la valeur et la solution associées à $(SP2_k)$

Concernant l'ajout de coupes sur le poids des clusters, nous avons fait le choix d'ajouter des contraintes uniquement pour les clusters n°k tels que le poids correspondant z_2^k dépasse B. Ce choix est à nuancer avec le cas où l'on rajoute l'ensemble des K contraintes associées à chacun des clusters.

2 Comparaison des résultats

Le tableau en annexe présente des résultats de performances obtenus à partir des implémentations sous Julia des différentes méthodes de résolution, sur les dix premières instances.

Nous nous sommes restreints à ces instances car aucune des méthodes (mis à part pour l'heuristique) n'a donné de résultats concluants en l'espace de dix minutes. Au-delà d'une vingtaine de noeuds, les temps de résolution étaient déjà trop longs pour continuer.

Les résultats montrent clairement que la méthode par dualisation donne les meilleurs résultats car elle a été capable de résoudre plus d'instances en une dizaine de secondes et donne de meilleures valeurs de l'objectif globalement.

Pour les trois dernières instances, l'objectif semble avoir été atteint (sauf peut-être pour la dernière instance) car peu d'améliorations ont été observées après quelques minutes. La résolution n'est pourtant pas achevée car le gap affiché décroît très lentement. Cela est sûrement dû à une mauvaise relaxation linéaire du problème dualisé.

En ce qui concerne branch & cut, les résultats sont moins bons car le programme met plus de temps à résoudre que pour "dualisation". C'est encore pire pour plans coupants qui ne parvient même pas à fournir de résultats pour les trois dernières instances.

Dualisation

```
using JuMP
using CPLEX

function fast_dualisation(inputFile::String, TimeLimit::Int64)

    include(inputFile)

    start = time()

    #definition de l
    l = [sqrt((coordinates[i, 1]-coordinates[j, 1])^2 + (coordinates[i, 2] - coordinates

    m = Model(CPLEX.Optimizer)

    #Variables primales
    @variable(m, x[i in 1:n, j in i+1:n], Bin)
    @variable(m, y[k in 1:K, i in k:n], Bin)

    for j in 1:n
        for i in j+1:n
            x[i, j] = x[j, i]
        end
    end

    #Variables duales
    @variable(m, beta[i in 1:n, j in i+1:n] >= 0)
    @variable(m, alpha >= 0)
    @variable(m, gamma[k in 1:K] >= 0)
    @variable(m, zeta[k in 1:K, i in k:n] >= 0)

    for j in 1:n
        for i in j+1:n
                 beta[i, j] = beta[j, i]
        end
    end


    #Contraintes Xcomb
    @constraint(m, [k in 1:K, i in k:n, j in i+1:n], y[k,i] + y[k,j] <= 1+x[i,j])

    @constraint(m, [i in 1:K], sum(y[k,i] for k in 1:i) == 1)
    @constraint(m, [i in K+1:n], sum(y[k,i] for k in 1:K) == 1)

    @constraint(m, [k in 1:K], sum(y[k,i] for i in k:n) >= 1)

    #Contraintes Xnum et autres dualisees
    @constraint(m, [i in 1:n, j in i+1:n], alpha + beta[i,j] >= (lh[i] + lh[j])*x[i,j])
    @constraint(m, [k in 1:K], W*gamma[k] + sum(w_v[i]*y[k,i] + W_v[i]*zeta[k,i] for i i

    @constraint(m, [k in 1:K, i in k:n], gamma[k] + zeta[k,i] >= w_v[i]*y[k,i])


    #Objectif
    @objective(m, Min, L*alpha + sum(l[i,j]*x[i,j] + 3*beta[i,j] for i in 1:n,j in i+1:n
```

```
    # Definit une limite de temps
    set_optimizer_attribute(m, "CPX_PARAM_TILIM", TimeLimit)

    #Resolution
    optimize!(m)

    computation_time = time() - start

    # Recuperation du status de la resolution
    feasibleSolutionFound = primal_status(m) == MOI.FEASIBLE_POINT
    isOptimal = termination_status(m) == MOI.OPTIMAL
    if feasibleSolutionFound
        # Recuperation de la valeur de l'objectif
        vOpt = JuMP.objective_value(m)
        vy = JuMP.value.(y)
    else
        vOpt = -1
    end

    vy_conv = zeros(K,n)
    for k in 1:K
        for i in k:n
            vy_conv[k,i] = vy[k,i]
        end
    end

    Clusters = []
    for k in 1:K
        push!(Clusters, findall(!iszero, vy_conv[k,:]))
    end

    return vOpt, computation_time, Clusters
end
```

Branch & cut

2.1 Résolution des sous-problèmes $(SP1)$ et $(SP2_k)$

```
using JuMP
using CPLEX

function fast_SP1_solve(inputFile::String, x_val, l, TimeLimit::Int64)

    include(inputFile)

    #Sous-probleme 1
    SP1 = Model(CPLEX.Optimizer)

    @variable(SP1, delta1[i in 1:n, j in i+1:n] >= 0)

    @constraint(SP1, sum(delta1[i,j] for i in 1:n,j in i+1:n) <= L)
    @constraint(SP1, [i in 1:n, j in i+1:n], delta1[i,j] <= 3)

    @objective(SP1, Max, sum((l[i,j] + delta1[i,j]*(lh[i]+lh[j]))*x_val[i,j]
for i in 1:n,j in i+1:n))

    # Desactive les sorties de CPLEX (optionnel)
    set_optimizer_attribute(SP1, "CPX_PARAM_SCRIND", 0)

    # Definit une limite de temps
    set_optimizer_attribute(SP1, "CPX_PARAM_TILIM", TimeLimit)

    #R solution
    optimize!(SP1)

    # Recuperation du status de la resolution
    feasibleSolutionFound = primal_status(SP1) == MOI.FEASIBLE_POINT
    isOptimal = termination_status(SP1) == MOI.OPTIMAL
    if feasibleSolutionFound
            # Recup ration des valeurs  d une  variable
            delta1_val = JuMP.value.(delta1)
            z1 = JuMP.objective_value(SP1)
    end

    # Conversion de delta1_val car type pas approprie pour plans_coupants
    delta1_val_converted = zeros(n,n)

    for i in 1:n
        for j in i+1:n
            delta1_val_converted[i,j] = delta1_val[i,j]
        end
    end
    return z1, delta1_val_converted
end

function fast_SP2k_solve(inputFile::String, y_val, TimeLimit::Int64)

    include(inputFile)

    #Sous-probleme 2,k
    delta2_val = Matrix{Float64}(undef,K,n)
```

```
    z2 =[]

    for k in 1:K
        SP2k = Model(CPLEX.Optimizer)

        @variable(SP2k, delta2k[i in k:n] >= 0)

        @constraint(SP2k, sum(delta2k[i] for i in k:n) <= W)
        @constraint(SP2k, [i in k:n], delta2k[i] <= W_v[i])

        @objective(SP2k, Max, sum(w_v[i]*(1 + delta2k[i])*y_val[k,i]  for i in k:n))

        # Desactive les sorties de CPLEX (optionnel)
        set_optimizer_attribute(SP2k, "CPX_PARAM_SCRIND", 0)

        # Definit une limite de temps
        set_optimizer_attribute(SP2k, "CPX_PARAM_TILIM", TimeLimit)

        #Resolution
        optimize!(SP2k)

        # Recuperation du status de la resolution
        feasibleSolutionFound = primal_status(SP2k) == MOI.FEASIBLE_POINT
        isOptimal = termination_status(SP2k) == MOI.OPTIMAL
        if feasibleSolutionFound
                # R cup ration des valeurs  d une  variable
                delta2_val[k,k:end] = JuMP.value.(delta2k)
                push!(z2, JuMP.objective_value(SP2k))
        end
    end

    return z2, delta2_val
end

using JuMP
using CPLEX

include("fast_sp_solving.jl")

function fast_branch_and_cut(inputFile::String, TimeLimit::Int64)

    include(inputFile)

    start = time()

    #definition de l
    l = [sqrt((coordinates[i, 1]-coordinates[j, 1])^2 + (coordinates[i, 2] - coordinates

    m = Model(CPLEX.Optimizer)

    # Il est impose d'utiliser 1 seul thread en Julia avec CPLEX pour
    # utiliser les callbacks
    MOI.set(m, MOI.NumberOfThreads(), 1)

    #Variables
    @variable(m, z >= 0)
    @variable(m, x[i in 1:n, j in i+1:n], Bin)
```

```
    @variable(m, y[k in 1:K, i in k:n], Bin)

    #Contraintes Xcomb
    @constraint(m, z >= sum(l[i,j]*x[i,j] for i in 1:n,j in i+1:n))

    @constraint(m, [k in 1:K, i in k:n, j in i+1:n], y[k,i] + y[k,j] <= 1+x[i,j])

    @constraint(m, [i in 1:K], sum(y[k,i] for k in 1:i) == 1)
    @constraint(m, [i in K+1:n], sum(y[k,i] for k in 1:K) == 1)

    @constraint(m, [k in 1:K], sum(w_v[i]*y[k,i] for i in k:n) <= B)

    #Objectif
    @objective(m, Min, z)

    #Definition du Callback
    function mon_super_callback(cb_data::CPLEX.CallbackContext, context_id::Clong)
        if isIntegerPoint(cb_data,context_id)

            CPLEX.load_callback_variable_primal(cb_data, context_id)

            # On recupere la valeur de x, y et de z
            x_val = callback_value.(cb_data, x)
            y_val = callback_value.(cb_data, y)
            z_val = callback_value(cb_data, z)

            # Resolution de SP1 et recuperation des donn es
            z1, delta1_val = fast_SP1_solve(inputFile, x_val, l, TimeLimit)

            if z1 > z_val
                cstr = @build_constraint(z >= sum((l[i,j] + delta1_val[i,j]*(lh[i]+lh[j]
for i in 1:n,j in i+1:n))
                MOI.submit(m, MOI.LazyConstraint(cb_data), cstr)
            end

            # Resolution de SP2_k et recuperation des donnees
            z2, delta2_val = fast_SP2k_solve(inputFile, y_val, TimeLimit)

            for k in 1:K
                if z2[k] > B
                    cstr = @build_constraint(sum(w_v[i]*(1 + delta2_val[k,i])*y[k,i]
for i in k:n) <= B)
                    MOI.submit(m, MOI.LazyConstraint(cb_data), cstr)
                end
            end
        end
    end

    # Definit une limite de temps
    set_optimizer_attribute(m, "CPX_PARAM_TILIM", TimeLimit)

    # On precise que le modele doit utiliser notre fonction de callback
    MOI.set(m, CPLEX.CallbackFunction(), mon_super_callback)
    optimize!(m)

    computation_time = time() - start
```

```
    # Recuperation du status de la resolution
    feasibleSolutionFound = primal_status(m) == MOI.FEASIBLE_POINT
    isOptimal = termination_status(m) == MOI.OPTIMAL
    if feasibleSolutionFound
        # Recuperation de la valeur de l'objectif
        vOpt = JuMP.objective_value(m)
        vy = JuMP.value.(y)
    else
        vOpt = -1 # No feasible solution found
    end

    vy_conv = zeros(K,n)

    for k in 1:K
        for i in k:n
            vy_conv[k,i] = vy[k,i]
        end
    end

    Clusters = []
    for k in 1:K
        push!(Clusters, findall(!iszero, vy_conv[k,:]))
    end

    return vOpt, computation_time, Clusters
end

# Fonction  permettant  de  determiner  si  c'est  l'obtention  d'une
# solution entiere qui a entraine l'appel d'un callback
function isIntegerPoint(cb_data::CPLEX.CallbackContext, context_id::Clong)

    # context_id  == CPX_CALLBACKCONTEXT_CANDIDATE si le  callback est
    # appele dans un des deux cas suivants :
    # cas 1 - une solution entiere a ete obtenue; ou
    # cas 2 - une relaxation non bornee a ete obtenue
    if context_id != CPX_CALLBACKCONTEXT_CANDIDATE
        return false
    end

    # Pour determiner si on est dans le cas 1 ou 2, on essaie de recuperer la
    # solution entiere courante
    ispoint_p = Ref{Cint}()
    ret = CPXcallbackcandidateispoint(cb_data, ispoint_p)

    # S'il n'y a pas de solution entiere
    if ret != 0 || ispoint_p[] == 0
        return false
    else
        return true
    end
end
```

Plans coupants

```
using JuMP
using CPLEX
include("fast_sp_solving.jl")

function fast_plans_coupants(inputFile::String, TimeLimit::Int64)

    include(inputFile)

    start = time()
    remaining_time = trunc(Int, TimeLimit - (time() - start))

    #definition de l
    l = [sqrt((coordinates[i, 1]-coordinates[j, 1])^2 + (coordinates[i, 2] - coordinates

    U1 = [l]
    U2 = [convert(Vector{Float64}, w_v)]

    # Premi re iteration

    m = Model(CPLEX.Optimizer)

    #Variables
    @variable(m, z >= 0)
    @variable(m, x[i in 1:n, j in i+1:n], Bin)
    @variable(m, y[k in 1:K, i in k:n], Bin)

    #Contraintes Xcomb
    @constraint(m, [l1 in U1], z >= sum(l1[i,j]*x[i,j] for i in 1:n,j in i+1:n))

    @constraint(m, [k in 1:K, i in k:n, j in i+1:n], y[k,i] + y[k,j] <= 1+x[i,j])

    @constraint(m, [i in 1:K], sum(y[k,i] for k in 1:i) == 1)
    @constraint(m, [i in K+1:n], sum(y[k,i] for k in 1:K) == 1)

    @constraint(m, [k in 1:K, w2_v in U2], sum(w2_v[i]*y[k,i] for i in k:n) <= B)

    #Objectif
    @objective(m, Min, z)

    # Desactive les sorties de CPLEX (optionnel)
    set_optimizer_attribute(m, "CPX_PARAM_SCRIND", 0)

    # Definit une limite de temps
    set_optimizer_attribute(m, "CPX_PARAM_TILIM", TimeLimit)

    optimize!(m)

    # Recuperation du status de la resolution
    feasibleSolutionFound = primal_status(m) == MOI.FEASIBLE_POINT
    isOptimal = termination_status(m) == MOI.OPTIMAL
    if feasibleSolutionFound
            # Recuperation des valeurs  d une  variable
            x_val = JuMP.value.(x)
            y_val = JuMP.value.(y)
```

```
        z_val = JuMP.objective_value(m)
end


# Resolution de SP1 et recuperation des donnees
z1, delta1_val = fast_SP1_solve(inputFile, x_val, l, TimeLimit)

# Actualisation de U1
if z1 > z_val
    l1 = l .+ delta1_val .*(lh .+ transpose(lh))
    push!(U1, l1)
end

# Resolution de SP2_k et recuperation des donnees
z2, delta2_val = fast_SP2k_solve(inputFile, y_val, TimeLimit)

# Actualisation de U2
if maximum(z2) > B
    for k in 1:K
        w2 = w_v .* (delta2_val[k,:] .+ 1)
        push!(U2, w2)
    end
end

# Nouvelles iterations de resolution du probleme maitre

while (z1 > z_val || maximum(z2) > B) && time() - start < TimeLimit

    m = Model(CPLEX.Optimizer)

    #Variables
    @variable(m, z >= 0)
    @variable(m, x[i in 1:n, j in i+1:n], Bin)
    @variable(m, y[k in 1:K, i in k:n], Bin)

    #Contraintes Xcomb
    for l1 in U1
        @constraint(m, z >= sum(l1[i,j]*x[i,j] for i in 1:n,j in i+1:n))
    end

    @constraint(m, [k in 1:K, i in k:n, j in i+1:n], y[k,i] + y[k,j] <= 1+x[i,j])

    @constraint(m, [i in 1:K], sum(y[k,i] for k in 1:i) == 1)
    @constraint(m, [i in K+1:n], sum(y[k,i] for k in 1:K) == 1)

    for w2 in U2
        @constraint(m, [k in 1:K], sum(w2[i]*y[k,i] for i in k:n) <= B)
    end

    #Objectif
    @objective(m, Min, z)

    # Desactive les sorties de CPLEX (optionnel)
    set_optimizer_attribute(m, "CPX_PARAM_SCRIND", 0)

    # Definit une limite de temps
```

```
        remaining_time = trunc(Int, TimeLimit - (time() - start))
        set_optimizer_attribute(m, "CPX_PARAM_TILIM", remaining_time)

        optimize!(m)

        # Recuperation du status de la resolution
        feasibleSolutionFound = primal_status(m) == MOI.FEASIBLE_POINT
        isOptimal = termination_status(m) == MOI.OPTIMAL
        if feasibleSolutionFound
                # Recuperation des valeurs  d une  variable
                x_val = JuMP.value.(x)
                y_val = JuMP.value.(y)
                z_val = JuMP.objective_value(m)
        end

        # Resolution de SP1 et recuperation des donnees
        z1, delta1_val = fast_SP1_solve(inputFile, x_val, l, TimeLimit)

        # Actualisation de U1
        if z1 > z_val
            l1 = l .+ delta1_val .*(lh .+ transpose(lh))
            push!(U1, l1)
        end

        # Resolution de SP2_k et recuperation des donnees
        z2, delta2_val = fast_SP2k_solve(inputFile, y_val, TimeLimit)

        # Actualisation de U2
        for k in 1:K
            if z2[k] > B
                w2 = w_v .* (delta2_val[k,:] .+ 1)
                push!(U2, w2)
            end
        end
    end

    computation_time = time() - start


    vy_conv = zeros(K,n)

    for k in 1:K
        for i in k:n
            vy_conv[k,i] = y_val[k,i]
        end
    end

    Clusters = []
    for k in 1:K
        push!(Clusters, findall(!iszero, vy_conv[k,:]))
    end

    return z1, computation_time, Clusters
end
```

Instance	Static sol		Robust Pb best value	Dualisation			Branch & Cut			Plans coupants			Heuristique		
	Value	Computation Time		Value	Computation Time	Gap	Value	Computation Time	Gap	Value	Computation Time	Gap	Value	Computation Time	Gap
10_ulysses_3.tsp	54,35	0,55	137,00	137,00	0,39	152%	137,00	1,35	152%	137	4,51	152%	136,09	5,083985658	150%
10_ulysses_6.tsp	7,22	0,38	55,12	55,12	0,54	663%	55,12	1,66	663%	55,12	4,89	663%	55,12	5,078963805	663%
10_ulysses_9.tsp	0,72	0,19	33,29	33,29	0,14	4524%	33,29	0,91	4524%	33,29	6,28	4524%	33,29	10,33714087	4524%
14_burma_3.tsp	66,21	0,84	93,39	93,39	0,52	41%	93,39	2,49	41%	93,39	4,56	41%	112,62	Exceed time	70%
14_burma_6.tsp	17,96	1,16	42,74	42,74	0,74	138%	42,74	2,87	138%	42,74	51,69	138%	42,74	5,873981856	138%
14_burma_9.tsp	4,72	0,80	20,76	20,76	2,64	340%	20,76	18,60	340%	20,76	179,4	340%	20,76	5,660169871	340%
22_ulysses_3.tsp	284,21	1,81	358,64	358,64	16,51	26%	358,64	43,79	26%	358,64	177,3	26%	309,80	5,610921881	9%
22_ulysses_6.tsp	82,63	37,69	116,53	116,53	Exceed 600s	41%	128,33	Exceed 600s	55%				90,63	Exceed time	10%
22_ulysses_9.tsp	26,22	8,98	64,97	64,97	Exceed 600s	148%	64,97	Exceed 600s	148%				46,95	Exceed time	79%
26_eil_3.tsp	1850,42	110,84	2421,65	2431,87	Exceed 600s	31%	2526,53	Exceed 600s	37%				2238,81	20,15336836	21%

Figure 1: Tables des performances sur les dix premières instances

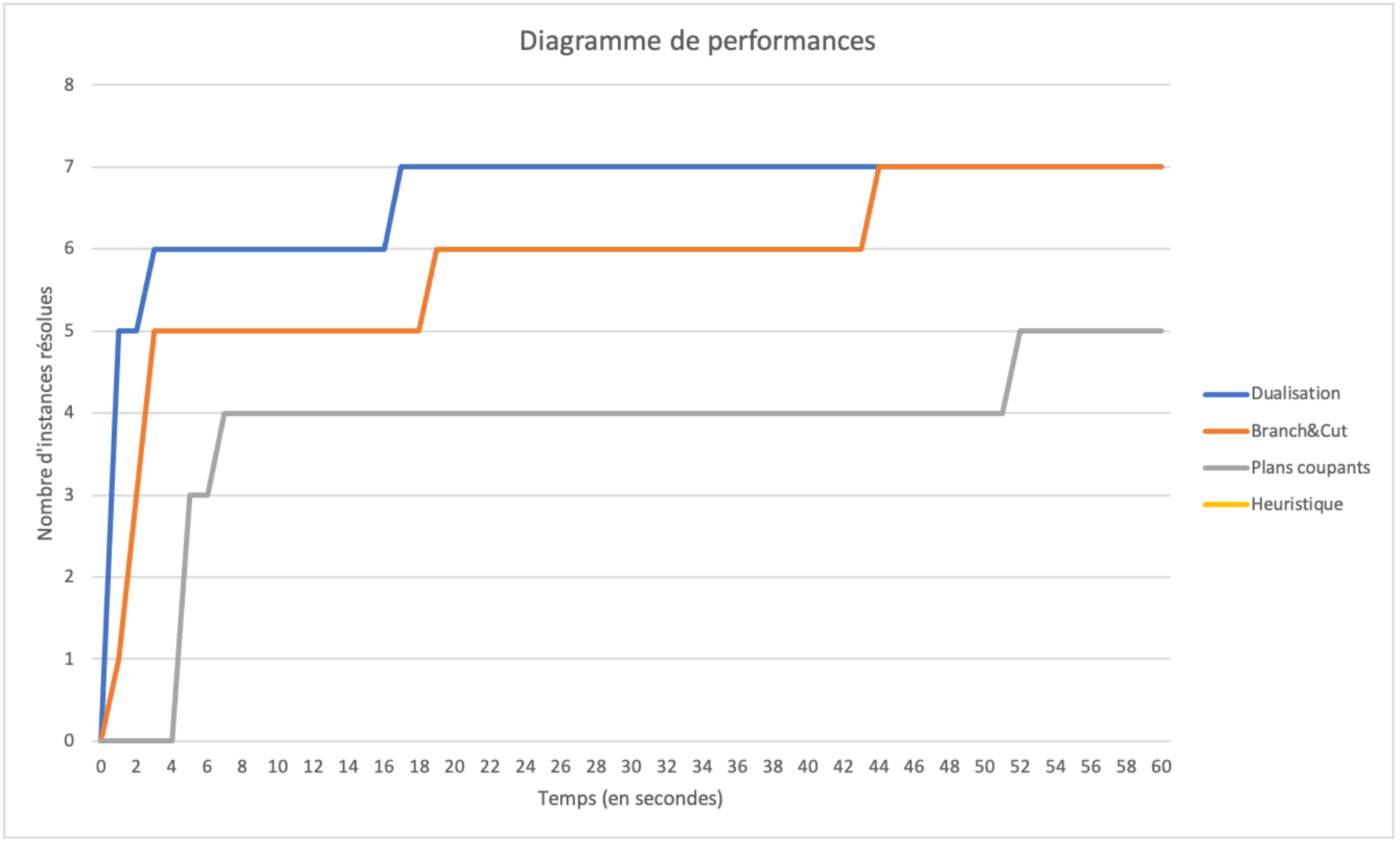

Figure 2: Diagramme de performance