Université de Montréal

Une heuristique de recherche à voisinage variable pour le problème du voyageur de commerce avec fenêtres de temps

Minimisation du temps du retour au dépôt

par Khalid Amghar

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2016

Résumé

Nous adaptons une heuristique de recherche à voisinage variable pour traiter le problème du voyageur de commerce avec fenêtres de temps (TSPTW) lorsque l'objectif est la minimisation du temps d'arrivée au dépôt de destination. Nous utilisons des méthodes efficientes pour la vérification de la réalisabilité et de la rentabilité d'un mouvement. Nous explorons les voisinages dans des ordres permettant de réduire l'espace de recherche. La méthode résultante est compétitive avec l'état de l'art. Nous améliorons les meilleures solutions connues pour deux classes d'instances et nous fournissons les résultats de plusieurs instances du TSPTW pour la première fois.

Mots-clés : recherche à voisinage variable, problème de voyageur de commerce, fenêtres de temps, minimisation du temps d'arrivée au dépôt, réalisabilité d'un mouvement, rentabilité d'un mouvement, réduction de l'espace de recherche.

Abstract

We adapt a general variable neighborhood search heuristic to solve the traveling salesman problem with time windows (TSPTW) where the objective is to minimize the completion time. We use efficient methods to check the feasibility and the profitability of a movement. We use a specific order to reduce the search space while exploring the neighborhoods. The resulting method is competitive with the state-of-the-art. We improve the best known solutions for two classes of instances and provide the results of multiple instances of TSPTW for the first time.

Keywords : general variable neighborhood search, traveling salesman problem, time windows, completion time minimization, feasibility, profitability, search space reduction.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles

ACO : *Ant Colony Optimization*

GVNS : *General Variable Neighborhood Search*

MPTW : *Makespan Problem with Time Windows*

NR : Non Rapporté

PRL : Procédure de Recherche Locale

SPEC : *Standard Performance Evaluation Corporation*

TSP : *Traveling Salesman Problem*

TSPTW : *Traveling Salesman Problem with Time Windows*

VND : *Variable Neighborhood Descent*

VNS : *Variable Neighborhood Search*

VRPTW : *Vehicle Routing Problem with Time Windows*

À mes chers parents, et à toutes les autres personnes que j'aime

Chapitre 1 : Introduction

Le problème du voyageur de commerce (TSP pour *Traveling Salesman Problem*) est un problème qui a été largement étudié par la communauté de la recherche opérationnelle. Plusieurs variantes de ce problème sont traitées dans la littérature (Ilavarasi et Joseph, 2014). Une des variantes les plus répandues de ce problème est le voyageur de commerce avec fenêtres de temps (TSPTW pour *Traveling Salesman Problem with Time Windows*). Ce problème a plusieurs applications, comme la livraison des colis, la gestion du transport scolaire ou du personnel d'une entreprise, l'ordonnancement de machines, etc.

Pour faciliter la définition du problème, nous l'exposons dans le cas d'une entreprise de livraison de colis : un véhicule chargé de colis doit partir d'un dépôt (on l'appellera par la suite dépôt origine) pour servir (ou visiter) un ensemble de n clients, puis revenir au dépôt (bien qu'il s'agisse du même dépôt, on appellera ce dépôt d'arrivée le dépôt destination). Chaque client doit être visité exactement une fois, et les clients doivent être visités dans un intervalle de temps (ou fenêtre de temps) connu à l'avance. Si le véhicule arrive à un client avant l'heure de début de cet intervalle, alors il doit attendre jusqu'à l'atteinte de l'heure de début pour servir ce client. Cependant, un client ne peut pas être servi si le véhicule arrive après l'heure de fin de la fenêtre de temps du client. Chaque client nécessite également un temps pour être servi. Le but est de choisir une tournée pour laquelle la valeur de la fonction objectif est la meilleure possible.

Pour avoir une idée de la difficulté du problème, on note qu'il existe $n!$ façons de réaliser une tournée qui visite chaque client une seule fois (pour le cas asymétrique, i.e., celui où le temps ou la distance nécessaire pour aller d'un client i vers un client j n'est pas forcément le même temps ou la même distance nécessaire pour aller du client j vers le client i). Ainsi, pour visiter 10 clients et revenir au dépôt il y a 3 628 800 choix (tournées) possibles, et pour visiter 50 clients, le nombre de tournées possibles est composé de 65 chiffres. L'introduction des fenêtres de temps joue un rôle double : d'une part, elle peut réduire considérablement le nombre des solutions réalisables, soit les tournées qui respectent les conditions citées dans la définition du problème; d'autre part, elle complique la tâche de trouver au moins une solution réalisable si une telle solution existe. En effet, Savelsbergh (1985) prouve que le problème de trouver une

solution réalisable pour le TSPTW est un problème fortement NP-complet même lorsque la matrice des temps (ou distances) est symétrique et qu'elle satisfait l'inégalité triangulaire.

Comme cité auparavant, le but est de choisir une tournée pour laquelle la valeur de la fonction objectif est la meilleure possible. La fonction objectif peut varier dépendamment de l'application et des priorités des décideurs. On peut viser, entre autres : la minimisation de la distance totale de la tournée; la minimisation du temps d'attente total de la tournée; la minimisation de la durée totale de la tournée, c'est-à-dire le temps d'arrivée au dépôt destination moins le temps de départ du dépôt origine (on rappelle qu'il s'agit bien du même dépôt, mais on ajoute les mots origine et destination pour faciliter la distinction de l'ordre de visite). Pour la variante du TSPTW correspondant à cette fonction objectif, le temps de départ du dépôt origine n'est pas connu à l'avance, il est lui-même une variable de décision. Un autre objectif possible est la minimisation du temps d'arrivée au dépôt destination. C'est cette dernière fonction objectif que nous avons considérée dans notre travail.

Notre travail s'inscrit dans le cadre d'un cas pratique d'une entreprise de développement de solutions logicielles intégrées pour la gestion du transport et de la livraison du courrier et des colis.

Nous présentons dans ce mémoire une méthode heuristique à deux phases basée sur la métaheuristique de recherche à voisinage variable, ou VNS pour *Variable Neighborhood Search* (Mladenović et Hansen 1997; Hansen, Mladenović et Pérez 2008). A la première phase, dite constructive, nous trouvons une solution réalisable à l'aide d'une heuristique de type VNS présentée par Da Silva et Urrutia (2010). A la deuxième phase, nous essayons d'améliorer la solution réalisable trouvée à la phase constructive. Nous utilisons une variante de VNS appelée GVNS (*General Variable Neighborhood Search*), une heuristique présentée par Mladenovic, Todosijevic et Urosevic (2012) qui s'est avérée robuste et efficiente en traitant le TSPTW où l'objectif est de minimiser la distance totale. Nous l'adaptons donc à la variante du TSPTW où l'objectif est de minimiser le temps d'arrivée au dépôt, en introduisant notamment une approche efficiente pour la vérification de la rentabilité d'un mouvement, et en la combinant à des méthodes existantes permettant la vérification de la réalisabilité d'un mouvement et la réduction du nombre de voisins visités. Nous discutons également l'ordre des voisinages à considérer, et pour chaque voisinage nous expliquons comment l'ordre de parcours joue un rôle crucial pour

l'accélération de l'exploration du voisinage. Nous introduisons également des améliorations au niveau technique permettant l'accélération de l'exploration du voisinage. Nous améliorons les meilleures solutions connues de deux classes d'instances. Nous fournissons, au mieux de nos connaissances, les résultats de plusieurs instances pour la première fois lorsque la fonction objectif est le temps d'arrivée au dépôt. La méthode présentée est robuste et les résultats obtenus sont compétitifs avec la méthode considérée comme l'état de l'art actuellement. En ce qui concerne le cas pratique des instances fournies par l'entreprise qui vise à améliorer son algorithme traitant le TSPTW, notre méthode améliore considérablement les résultats actuels de l'entreprise.

Nous présenterons dans le deuxième chapitre la motivation de ce travail. Nous rappellerons la définition du problème tout en introduisant les notations utilisées et nous discuterons également de quelques fonctions objectif. Nous présenterons plusieurs travaux de la littérature traitant du TSPTW.

Le troisième chapitre donnera une idée générale de la métaheuristique de recherche à voisinage variable VNS, ainsi que sa variante GVNS. Nous citerons également des voisinages et des mouvements de recherche locale connus pour le TSPTW, notamment ceux que nous utiliserons dans notre méthode.

L'algorithme que nous proposons, ainsi que les techniques utilisées pour son accélération seront présentés au quatrième chapitre.

Dans le cinquième chapitre, nous présenterons les résultats expérimentaux de notre méthode pour plusieurs instances connues dans la littérature, et nous analyserons ces résultats en les comparant, quand cela est possible, à des résultats existants. Nous présenterons également les résultats de notre méthode pour les instances du cas pratique que nous avons cité auparavant, et nous les comparons aux résultats actuels de l'entreprise en question.

Chapitre 2 : Définition du problème et revue de la littérature

2.1 Le problème traité

2.1.1 Motivation

Notre travail s'inscrit dans le cadre d'un cas pratique d'une entreprise de développement de solutions logicielles intégrées pour la gestion du transport et de la livraison du courrier et des colis.

Cette entreprise offre des solutions permettant de traiter notamment le problème de tournées de véhicules avec fenêtres de temps (VRPTW pour *Vehicle Routing Problem with Time Windows*). L'objectif est de minimiser la durée totale des tournées. L'entreprise juge avoir un outil très performant permettant de définir la tournée à réaliser par chaque véhicule et offrant également la possibilité de définir le temps de départ du dépôt de chaque véhicule (cette approche est connue sous le nom de *cluster-first route-second*). L'entreprise vise donc à améliorer l'algorithme traitant chaque tournée séparément comme un TSPTW. Nous exploitons le fait que l'outil de l'entreprise indique le temps de départ de chaque véhicule comme suit : étant donné un temps de départ connu, la minimisation de la durée d'une tournée revient à minimiser le temps d'arrivée au dépôt destination.

Un facteur à prendre en considération est la diversité des tournées à traiter. L'algorithme à proposer doit être capable de traiter des instances très variées : un nombre de clients variant d'une instance à l'autre; des fenêtres de temps de largeurs différentes ou encore des clients avec des fenêtres de temps et d'autres non. Les temps d'exécution doivent être raisonnables (inférieurs à une minute dans tous les cas) et les solutions obtenues doivent être de bonne qualité. Nous pouvons déduire que la robustesse de l'algorithme sera importante pour ce genre d'applications.

2.1.2 Position du problème et notations utilisées

Nous rappelons la définition du problème du voyageur de commerce avec fenêtres de temps, ou TSPTW, en indiquant d'abord la notation que nous utiliserons par la suite :

- n est le nombre de clients;
- N^C est l'ensemble des clients;
- g est le dépôt origine;
- h est le dépôt destination (en réalité, g et h représentent le même dépôt);
- $G=(N, R)$ est un graphe orienté;
- N est l'ensemble des nœuds de $G=(N, R)$;
- $N = N^C \cup \{g, h\}$;
- R est l'ensemble des arcs de $G=(N, R)$;
- T est la matrice des temps ou des coûts;
- $t_{i,j}$ est le temps ou le coût nécessaire pour se déplacer du nœud i au nœud j (ce temps inclut également le temps de service au nœud i, qui est nul pour le dépôt);
- A_i est le temps d'arrivée au nœud i;
- D_i est le temps du départ du nœud i;
- a_i est le temps du début de la fenêtre de temps du nœud i;
- b_i est le temps de fin de la fenêtre de temps du nœud i;
- w_i est le temps d'attente avant de commencer le service du nœud i (ce temps d'attente est nul pour le dépôt);
- TSPTW-T : le TSPTW lorsque l'objectif est de minimiser la distance totale (*Travel time*) de la tournée;

- TSPTW-D : le TSPTW lorsque l'objectif est de minimiser la durée totale de la tournée;
- TSPTW-C : le TSPTW lorsque l'objectif est de minimiser le temps d'arrivée au dépôt destination (*Completion time*).

Nous rappelons que l'attente est permise si on arrive à un client avant a_i, c'est-à-dire avant le début de la fenêtre de temps du client i. Ainsi, on a les relations suivantes :

- $D_i = max\{A_i, a_i\}$;
- $w_i = max\{0, a_i - A_i\}$.

Le problème du voyageur de commerce avec fenêtres de temps TSPTW consiste donc à trouver une tournée qui commence au dépôt g, passe par tous les clients de l'ensemble N^C exactement une fois tout en respectant les contraintes sur les fenêtres de temps, et retourne au dépôt h (nous rappelons que g et h sont le même dépôt en réalité). L'objectif est de choisir une tournée de coût minimal. Nous discuterons par la suite quelques fonctions objectif possibles et pratiques pour le TSPTW.

2.1.3 La fonction objectif

Plusieurs objectifs sont possibles pour le TSPTW. Nous en citons trois :

- La minimisation de la distance totale de la tournée, soit le temps passé dans les déplacements entre les nœuds, *travel time*. La valeur de la fonction objectif ne tient pas compte des temps d'attente, elle tient compte uniquement de la somme des temps nécessaires pour les déplacements. Cette fonction objectif peut être utile dans les cas, entre autres, où l'on vise la minimisation de la consommation du carburant, ou lorsque l'on veut éviter l'usure des véhicules. Nous remarquons que cette variante (TSPTW-T) est la plus étudiée dans la littérature du TSPTW.

- La minimisation de la durée totale de la tournée, soit le temps d'arrivée au dépôt destination moins le temps de départ du dépôt origine. La valeur de la fonction objectif tient compte des temps d'attente. Dans cette variante (TSPTW-D), le temps de départ du dépôt origine n'est pas fixé à l'avance, il est une variable de décision. Cette variante peut être utile lorsque l'on veut minimiser les charges liées au temps d'utilisation de la source : la source peut être un véhicule, et l'une des charges peut être le salaire du chauffeur, ou encore la source peut être une machine quelconque dans le domaine de l'ordonnancement des tâches sur une machine que l'on veut utiliser le moins possible.
- La minimisation du temps d'arrivée au dépôt destination. La valeur de la fonction objectif tient compte des temps d'attente. Dans cette variante (TSPTW-C), le temps de départ du dépôt est généralement fixé au début de la fenêtre de temps du dépôt. Cette variante peut être utile lorsque les temps de départ des véhicules sont fixés à l'avance, notamment dans le cas du transport scolaire et du transport du personnel d'une entreprise, et que l'on veut que les véhicules reviennent le plus tôt possible pour que les étudiants ou les employés ne soient pas en retard. Cette variante peut être également intéressante dans le domaine de l'ordonnancement des tâches sur une machine lorsque le but est que la machine exécute toutes les tâches le plus tôt possible pour que cette machine soit libre et prête à exécuter un autre ensemble de tâches. C'est cette variante (TSPTW-C) que nous avons considérée dans notre travail. Cette variante est aussi connue sous le nom de MPTW, pour *Makespan Problem with Time Windows*.

En parcourant la littérature, nous avons remarqué quelques auteurs justifiant le choix d'une fonction objectif par rapport à d'autres en disant qu'elle est plus réaliste que les autres. Cependant, nous pensons que toute fonction objectif peut être réaliste et intéressante dépendamment du contexte de son application. Nous soulignons également que pour un même type d'application, la fonction objectif peut différer d'un pays à l'autre, car le coût prépondérant peut être le salaire horaire des chauffeurs pour un pays, alors que le coût prépondérant peut être

lié à l'usure du véhicule pour un autre pays. En effet, le décideur doit prendre plusieurs facteurs en considération avant de choisir la fonction objectif la mieux adaptée.

2.2 Revue de la littérature

La littérature concernant le TSPTW est tellement riche qu'il est très compliqué d'en faire une revue exhaustive. Ainsi, nous allons nous contenter de citer quelques travaux passés.

Nous soulignons que même si nous traitons la variante TSPTW-C, nous allons présenter des travaux de la littérature des autres variantes également, notamment le TSPTW-T et le TSPTW-D. Ceci se justifie par le fait que même si les fonctions objectif sont différentes, la structure du problème reste pratiquement la même.

Nous présenterons d'abord des méthodes exactes de la littérature pour le TSPTW, suivies de méthodes heuristiques.

2.2.1 Méthodes exactes

Christofides, Mingozzi et Toth (1981) résolvent le TSPTW-C en utilisant une combinaison de *branch-and-bound* et de la programmation dynamique. En fait, l'espace d'état à une itération donnée est relaxé de telle sorte à pouvoir déduire une borne inférieure. Cette borne est alors utilisée dans un algorithme de type *branch-and-bound*. Les auteurs discutent également des techniques permettant d'améliorer la borne inférieure obtenue. Les auteurs rapportent les résultats d'instances contenant jusqu'à 50 nœuds.

Baker (1983) résout le TSPTW-C en utilisant un modèle contenant un seul type de variables. Ces variables temporelles indiquent le temps où le voyageur de commerce visite un certain nœud. L'auteur commence d'abord par relaxer quelques contraintes, puis écrit le dual du sous-problème relaxé correspondant. Ce dual est en fait un problème de plus long chemin dans un graphe orienté. Il introduit par la suite les contraintes qu'il avait relaxées en expliquant les changements entrainés dans le dual. La résolution du dual procure de l'information concernant la borne inférieure du problème primal. Cette borne est utilisée par un algorithme

branch-and-bound. Baker (1983) utilise également quelques techniques simples, dans une étape de prétraitement, pour éliminer les arcs incompatibles (un arc incompatible est un arc qui ne peut pas exister dans une solution réalisable). Il résout des instances contenant jusqu'à 50 clients.

Langevin et al. (1993) présentent un modèle qui permet de résoudre le TSPTW-T et le TSPTW-C (il peut être généralisé pour traiter plusieurs variantes du TSP). Ils résolvent leur problème par une approche *branch-and-bound*. Le modèle présenté est sous forme d'un problème de flot à deux produits. Les auteurs expliquent comment leur modèle traite les contraintes de fenêtres de temps. Le modèle est ensuite relaxé pour dériver une borne inférieure. Cette borne peut être améliorée grâce à l'ajout des contraintes d'élimination des sous-tours qui sont violées. Ils présentent également plusieurs procédures d'élimination des arcs non compatibles (les arcs qui ne peuvent pas exister dans une solution réalisable) et de réduction des fenêtres de temps. Ces procédures sont appliquées itérativement jusqu'à ce qu'il n'y ait plus d'amélioration (élimination d'un arc ou réduction d'une fenêtre de temps) possible. Les dites procédures sont appliquées à chaque nœud de l'arbre de *branch-and-bound*, et non pas juste dans l'étape du prétraitement comme dans Baker (1983). Les auteurs présentent des résultats numériques pour des instances contenant jusqu'à 60 clients.

Dumas et al. (1995) considèrent le TSPTW-T. Ils introduisent de nouveaux tests d'élimination permettant d'améliorer la performance d'une approche de programmation dynamique relativement bien établie. Ces tests permettent, grâce aux contraintes des fenêtres de temps, de réduire l'espace d'états et le nombre des transitions d'états. Les auteurs ramènent le problème à la résolution d'un problème de plus court chemin dans un graphe où les nœuds représentent les états, et les arcs sont les transitions d'un état à un autre. Ils rapportent des résultats pour des instances contenant jusqu'à 200 clients. Les auteurs remarquent également que même si le TSPTW est un cas particulier du VRPTW (VRPTW avec un seul véhicule et sans contrainte sur la capacité du véhicule), on ne peut pas prétendre que chaque méthode qui fonctionne bien pour le VRPTW aura le même comportement pour le TSPTW (par exemple, il peut y avoir des problèmes de dégénérescence).

Pesant et al. (1998) formulent le TSPTW-T de manière adaptée à la programmation par contraintes, puis le résolvent par *branch-and-bound*. L'utilisation des techniques d'élimination d'arcs incompatibles et de réduction des fenêtres de temps n'est pas restreinte à la phase de

prétraitement et elle est réalisée de manière dynamique au cours de l'exécution de l'algorithme. Ces techniques peuvent être vues comme des contraintes redondantes ajoutées à leur modèle de base (de programmation par contraintes).

Focacci, Lodi et Milano (2002) présentent un algorithme exact pour la résolution du TSPTW-T. Ils utilisent la programmation par contraintes pour traiter les contraintes des fenêtres de temps. L'optimisation est guidée par un arbre de recherche (semblable au *branch-and-bound*) qui exploite, entre autres, des coupes pour trouver des bornes inférieures permettant d'élaguer des nœuds qui ne mèneront pas à des solutions optimales.

Li (2009) utilise une approche bidirectionnelle de la programmation dynamique à ressources limitées pour résoudre le TSPTW-T. On imagine le dépôt comme étant deux dépôts différents, un dépôt origine et un dépôt destination. On commence l'extension des étiquettes en partant en même temps du dépôt origine vers le dépôt destination, et du dépôt destination vers le dépôt origine. Comme les "ressources" sont limitées, chaque sens de parcours du graphe ne peut pas utiliser plus que la moitié des ressources. La connexion entre les étiquettes est établie alors si les conditions de réalisabilité sont toutes respectées. La méthode proposée dépend plus de la largeur des fenêtres de temps que du nombre des clients du problème. Li (2009) cite par exemple le fait que sa méthode a pu résoudre de façon optimale 4 parmi 5 instances -des instances proposées par Dumas et al. (1995)- contenant 100 clients avec une largeur allant jusqu'à 80 unités pour les fenêtres de temps, tandis qu'elle n'a pu résoudre aucune instance contenant 80 clients avec des fenêtres de temps allant jusqu'à 100 unités.

Baldacci, Mingozzi et Roberti (2012) résolvent le TSPTW-T avec une approche de programmation dynamique basée sur trois relaxations du problème. Deux relaxations existaient déjà dans la littérature, alors que la troisième a été introduite dans leur article et est appelée ngL-tour. Ils rapportent les résultats de 270 instances connues dans la littérature. Leur algorithme a pu résoudre 269 de ces instances. De plus, 136 de ces instances ont été résolues de manière optimale pour la première fois dans la littérature. Baldacci et al. (2012) affirment que leur méthode est meilleure que toutes les méthodes précédentes de la littérature.

Kara et al. (2013) traitent le TSPTW-D. Nous rappelons que la fonction objectif de cette variante est le temps d'arrivée au dépôt moins le temps de départ du dépôt et que le temps de

départ du dépôt est une variable de décision. Un modèle de programmation linéaire en nombres entiers est alors proposé, il contient O(n^2) variables binaires et O(n^2) contraintes, où n est le nombre de clients du problème. Ils résolvent le modèle directement en utilisant un solveur commercial, et rapportent les résultats obtenus pour des instances contenant jusqu'à 40 clients. Kara et al. (2013) affirment que leur modèle est meilleur que les modèles révisés de Langevin et al. (1993) et Desrosiers et al. (1995) en termes de la borne inférieure déduite par la relaxation linéaire du modèle, et en termes du temps de résolution, en utilisant directement un solveur commercial. Les auteurs indiquent que leur modèle peut être généralisé pour traiter le cas de plusieurs véhicules (VRPTW).

Nous avons présenté quelques méthodes exactes de la littérature concernant le TSPTW. Dépendamment de l'application, un décideur peut avoir besoin d'une solution offrant un bon ratio qualité-temps d'exécution, notamment dans le cas où le TSPTW est vu comme un problème de niveau opérationnel. Les limitations des méthodes exactes -ou bien on n'arrive pas à résoudre le problème ou bien la résolution nécessite un temps jugé non raisonnable par le décideur- ont poussé les chercheurs à développer des méthodes heuristiques. Ces heuristiques trouvent, en général, des solutions de bonne qualité mais ne fournissent aucune garantie quant à l'optimalité des solutions trouvées. Nous allons présenter quelques méthodes heuristiques dans la prochaine section.

2.2.2 Méthodes heuristiques

Savelsbergh (1985) a introduit une méthode basée sur la recherche locale pour traiter le TSPTW. Il prouve que le problème de trouver une solution réalisable pour le TSPTW est un problème fortement NP-complet, même lorsque la matrice des distances (ou temps) est symétrique et qu'elle satisfait l'inégalité triangulaire. Il traite le TSPTW-T et le TSPTW-C. Son algorithme est constitué de deux phases. Lors de la première phase, il trouve une solution réalisable, puis essaie d'améliorer cette solution dans la deuxième phase. Dans la première phase, Savelsbergh (1985) utilise une heuristique d'insertion dans l'unique but de trouver une solution réalisable. Cette heuristique insère les clients ayant des petites fenêtres de temps avant

ceux ayant des larges fenêtres de temps pour assurer la flexibilité de l'algorithme et avoir plus de choix pour les derniers clients à insérer. La deuxième étape de l'algorithme se charge de l'amélioration de la solution en appliquant des procédures de recherche locale. Il introduit notamment des méthodes efficientes de vérification de la réalisabilité et de la rentabilité. Pour la minimisation du temps de fin de la tournée, la rentabilité est par rapport à un objectif partiel. En minimisant cet objectif partiel, on est sûr que la valeur de la vraie fonction objectif ne peut pas se détériorer, mais on n'est pas sûr qu'elle s'améliore strictement. Nous notons que la méthode présentée par Savelsbergh (1985) n'utilise pas des tests d'élimination des arcs incompatibles.

Savelsbergh (1992) traite le cas du TSPTW où l'objectif est la minimisation de la durée de la tournée (TSPTW-D). Il utilise des techniques semblables à celles utilisées dans Savelsbergh (1985) (nous rappelons que ce dernier traite le TSPTW-C) et les adapte à la variante qui minimise la durée de la tournée TSPTW-D. Savelsbergh (1992) introduit notamment une méthode efficiente permettant de quantifier l'impact d'un mouvement sur la valeur de la fonction objectif. Nous notons là aussi que les tests d'élimination des arcs incompatibles -des arcs conduisant à des solutions non réalisables- ne sont pas utilisés dans cette méthode.

Carlton et Barnes (1996) résolvent le TSPTW-C et le TSPTW-T à l'aide d'une méthode de recherche tabou réactive (qui ajuste des paramètres au cours de l'exécution de l'algorithme). Ils utilisent une structure de hachage à deux niveaux qui permet d'avoir une très petite probabilité de collisions (une collision fait allusion au fait qu'une fonction de hachage détermine que deux solutions sont identiques -en comparant juste des parties des deux solutions- alors qu'elles sont différentes) sans être gourmande en termes de temps de calcul et de mémoire. Leur algorithme permet de visiter des solutions non réalisables en introduisant une pénalité statique dans la fonction objectif. Les auteurs indiquent que l'évaluation de la fonction objectif s'avère être gourmande en termes de temps de calcul. Carlton et Barnes (1996) considèrent en fait la minimisation de la distance (TSPTW-T), la minimisation du temps de fin de la tournée (TSPTW-C) et la minimisation d'une fonction objectif qui priorise le temps de fin de la tournée, puis choisit parmi les solutions qui ont le meilleur temps de fin celle ayant la distance totale la plus petite.

Gendreau et al. (1998) présentent une heuristique d'insertion pour le TSPTW-T. Cette approche essaie d'insérer les clients un à un en optimisant le tour courant à chaque itération tout en maintenant la réalisabilité de la solution. Si l'algorithme arrive à construire une solution réalisable, une procédure est alors exécutée pour essayer d'améliorer cette solution.

Favaretto, Moretti et Pellegrini (2006) optent pour une procédure basée sur l'optimisation par colonies de fourmis. Leur méthode est appliquée au TSPTW-T et au TSPTW-C. La première étape de leur algorithme essaie de trouver une solution réalisable à l'aide d'un algorithme basé sur le plus proche voisin, présenté par Solomon (1987). Toutefois, ce dernier algorithme n'assure pas que la solution initiale trouvée contient tous les clients. Ceci ne constitue pas un problème pour la méthode de Favaretto et al. (2006), car le principal but de cette première étape est d'avoir une idée approximative des coûts des solutions pour pouvoir déterminer la quantité initiale de "phéromone" sur les arcs. Dans la deuxième étape, l'algorithme de colonies de fourmis est alors exécuté en priorisant lors de la comparaison entre différentes solutions celle qui contient le plus de clients (nous rappelons que la première étape ne garantissait pas l'insertion de tous les clients).

Ohlmann et Thomas (2007) résolvent le TSPTW-T avec une heuristique basée sur le recuit comprimé (*Compressed-annealing*) qui est une variante du recuit simulé. Pour le TSPTW, les méthodes basées sur le recuit simulé permettent, avec une probabilité calculée à l'aide d'un paramètre appelé température, de faire un mouvement vers des solutions réalisables qui n'améliorent pas la solution courante. Ceci permet d'échapper aux problèmes liés aux optima locaux. Pour le recuit comprimé, on ajoute un autre paramètre, appelé pression, qui contrôle la probabilité de faire un mouvement vers une solution non réalisable, tout en ajoutant un terme de pénalité variable à la fonction objectif. Ohlmann et Thomas (2007) expliquent que, généralement, le fait d'opter pour une pénalité variable pour le recuit comprimé améliore les performances d'un algorithme de recuit simulé optant pour une pénalité statique bien choisie.

López-Ibáñez et Blum (2010) proposent de résoudre le TSPTW-T avec une méthode hybride qui combine l'optimisation par colonies de fourmis ACO (*Ant Colony Optimization*) à une méthode de recherche en faisceau (*Beam*), la méthode résultante est connue sous le nom de Beam-ACO (Blum, 2005). La recherche en faisceau est une heuristique qui permet de parcourir un graphe (généralement un arbre) en n'explorant qu'un certain nombre des nœuds fils. Elle

peut être vue comme une version heuristique de *branch-and-bound*. La performance de la recherche en faisceau dépend fortement de la fonction utilisée pour évaluer la qualité d'une partie de la solution (dans le cas du TSPTW, une partie de la solution correspond à un chemin qui ne contient pas tous les clients du problème) et de la borne inférieure déduite pour réduire l'espace de recherche. Les auteurs présentent des techniques stochastiques d'échantillonnage pour avoir de l'information sur les bornes. La combinaison résultante de la recherche en faisceau avec la nature constructive des algorithmes de colonies de fourmis a donné de bons résultats pour le TSPTW-T.

Da Silva et Urrutia (2010) optent pour une approche basée sur une variante de la recherche à voisinage variable appelée GVNS (*General Variable Neighborhood Search*) pour résoudre le TSPTW-T. Leur méthode est composée de deux phases. La première phase, dite constructive, se charge de trouver une solution réalisable à l'aide d'une heuristique de recherche à voisinage variable (VNS pour *Variable Neighborhood Search*). Cette nouvelle méthode s'avère plus efficace que celles présentées dans Gendreau et al. (1998) et Calvo (2000) pour trouver une solution réalisable. La deuxième phase, dite d'amélioration, de la méthode de Da Silva et Urrutia (2010) se charge d'améliorer la solution trouvée dans la première partie. Elle utilise une variante de VNS nommée GVNS. Leur algorithme utilise également un prétraitement de base pour éliminer des arcs incompatibles (nous rappelons qu'un arc incompatible est un arc qui ne peut pas apparaitre dans une solution réalisable). Les résultats obtenus améliorent les meilleures solutions connues pour quelques instances.

Mladenovic et al. (2012) optent eux aussi pour une approche GVNS pour résoudre le TSPTW-T. Comme dans Da Silva et Urrutia (2010), leur méthode se déroule en deux étapes. La première étape utilise l'algorithme de la phase constructive présenté dans ce dernier article pour trouver une solution réalisable -nous rappelons que c'est une heuristique VNS. La deuxième étape de leur algorithme essaie, avec une heuristique GVNS, d'améliorer la solution réalisable trouvée à la première étape. Les principales différences de leur méthode par rapport à la méthode de Da Silva et Urrutia (2010) sont les suivantes : l'utilisation d'un vecteur permettant de vérifier de manière efficiente la réalisabilité d'un mouvement ; le parcours de plus de voisinages ; et la non-utilisation du prétraitement permettant de déterminer et éliminer les arcs incompatibles (ceux conduisant à des solutions non réalisables). Ceci pousse leur algorithme à

utiliser une autre façon de parcourir les voisinages, mais cette nouvelle façon ne permet pas d'être sûr que toutes les solutions réalisables sont visitées ou évaluées. Mladenovic et al. (2012) améliorent les meilleures solutions connues pour quelques instances.

López-Ibáñez et al. (2013) résolvent le TSPTW-C (nous rappelons que c'est cette variante qui nous intéresse dans ce document) en adaptant deux méthodes (*Compressed-annealing* et *Beam-ACO*) qui avaient donné de bons résultats pour le TSPTW-T (ils adaptent les méthodes de Ohlmann et Thomas (2007) et López-Ibáñez et Blum (2010)). Les auteurs indiquent que, pour résoudre le TSPTW-C, il est intéressant d'essayer d'abord d'adapter des méthodes qui ont démontré leur efficacité pour le TSPTW-T (en profitant notamment du fait que le TSPTW-T est la variante la plus traitée dans la littérature parmi les trois variantes présentées à la section 2.1.3). Pour chacune des deux méthodes, les auteurs exécutent un processus rigoureux pour choisir les meilleurs paramètres et configurations pour chaque groupe d'instances présenté dans leur article. Les résultats obtenus sont bons pour les deux méthodes, mais l'algorithme *Beam-ACO* est significativement meilleur que la méthode *CA*. Les auteurs améliorent les meilleures solutions connues pour quelques instances, et présentent pour la première fois, pour le TSPTW-C, les résultats de plusieurs groupes d'instances connus pour le TSPTW-T.

Yurtkuran et Emel (2014) utilisent une heuristique basée sur des principes électromagnétiques pour résoudre le TSPTW-T. Cette méthode utilise l'attraction et la répulsion des particules chargées dans un espace à plusieurs dimensions. Les auteurs introduisent notamment des techniques pour permettre à leur méthode de traiter les contraintes sur les fenêtres de temps, et pour limiter l'espace de recherche

Tel que mentionné auparavant, nous ne prétendons pas avoir fait une revue exhaustive de la littérature, mais nous avons essayé de donner une idée des méthodes exactes et heuristiques déjà utilisées pour résoudre le TSPTW. Pour le lecteur intéressé, quelques articles présentent des revues de la littérature, par exemple Solomon et Desrosiers (1988) et Kona, Burde et Zanwar (2015).

Chapitre 3 : La recherche à voisinage variable et quelques voisinages de recherche locale pour le TSPTW

3.1 La recherche à voisinage variable

Nous présentons dans ce chapitre quelques généralités sur la recherche à voisinage variable, ainsi que des procédures que nous jugeons importantes pour la compréhension de l'algorithme que nous proposons dans ce document.

Ce chapitre est fortement inspiré de Hansen et al. (2008). Nous utilisons également les mêmes noms de fonctions et de procédures dans le but de garder des noms significatifs permettant au lecteur de faire le lien facilement avec l'article cité, qui permet de définir un cadre général de la métaheuristique VNS.

La recherche à voisinage variable (VNS pour *Variable Neighborhood Search*) est une métaheuristique qui a été présentée par Mladenović et Hansen (1997). Elle est basée sur le changement de voisinage. Ce changement peut être effectué dans l'étape d'intensification, qui vise à trouver un minimum local, ainsi que dans l'étape de diversification, qui vise à échapper à un minimum local. On peut voir la recherche à voisinage variable comme un cadre général permettant de gérer l'ordre d'exécution de procédures de recherche locale.

La VNS est fondée selon trois idées principales : un minimum local par rapport à un certain voisinage n'est pas forcément un minimum local par rapport à un autre voisinage ; un minimum global est un minimum local par rapport à tous les voisinages possibles ; pour plusieurs problèmes, les minima locaux par rapport à un ou plusieurs voisinages sont relativement proches l'un de l'autre.

Les trois idées précédemment citées évoquent toutes la notion de minimum local, ce qui veut dire que la détermination d'un minimum local est essentielle pour la VNS. Une procédure de recherche locale est une procédure itérative permettant d'identifier, pour un point initial x, un minimum local par rapport à un voisinage $V(x)$. Deux types de procédures de recherche locale sont largement utilisés : la recherche suivant la direction de la plus forte descente (que nous

appellerons par la suite *best improvement*) ; et la recherche suivant la première direction de descente (que nous appellerons par la suite *first improvement*). Les algorithmes des deux types de procédures sont les suivants :

Procédure *BestImprovement(x)*;
1 **répéter**
2 $x' \leftarrow x$;
3 $x \leftarrow arg\ min_{y \in V(x)} f(y)$
Jusqu'à $\left(f(x) \geq f(x')\right)$;

Figure 1. Algorithme *best improvement*

Procédure *FirstImprovement(x)*;
1 **répéter**
2 $x' \leftarrow x;\ \ i \leftarrow 0;$
3 **répéter**
4 $i \leftarrow i + 1;$
5 $x \leftarrow arg\ min\{f(x), f(x_i)\},\ \ x_i \in V(x)$
jusqu'à $(f(x) < f(x_i)\ \ ou\ \ i = |V(x)|);$
jusqu'à $\left(f(x) \geq f(x')\right);$

Figure 2. Algorithme *first improvement*

Nous allons présenter par la suite une fonction essentielle pour la compréhension de l'algorithme de VNS. Cette fonction qu'on appelle *NeighborhoodChange* permet un changement systématique de voisinage. En effet, étant donné une solution 'actuelle' x ou un point actuel (*incumbent point*), dont la valeur de la fonction objectif est $f(x)$, et un nombre entier k, la fonction *NeighborhoodChange* compare la valeur $f(x)$ à la valeur $f(x')$ d'une autre solution x'. Si $f(x')<f(x)$ alors on réinitialise k ($k=1$) et x' devient la nouvelle solution 'actuelle', sinon on incrémente la valeur de k ($k=k+1$) et la solution 'actuelle' reste inchangée (i.e., x continue à être la solution 'actuelle'). Voici l'algorithme de la fonction *NeighborhoodChange* :

Procédure *NeighborhoodChange* (x, x', k);

1 **si** $f(x') < f(x)$ **alors**

2 $x \leftarrow x'$; $k \leftarrow 1$ /* Modifier la solution actuelle */;

sinon

3 $k \leftarrow k + 1$;

fin

Figure 3. Algorithme de changement de voisinage

Maintenant que nous avons les procédures nécessaires, nous pouvons définir l'algorithme de base de la recherche à voisinage variable. Notons que nous n'avons pas présenté un algorithme pour la fonction *shake()*, c'est une fonction de perturbation qui introduit un aspect aléatoire à la VNS. La fonction de perturbation peut être choisie dépendamment de l'application de l'algorithme et elle peut dépendre également d'un paramètre k. Notons également que, pour l'algorithme que nous présenterons de la procédure VNS, le critère d'arrêt est choisi comme étant une limite de temps t_{max}.

```
    Procédure VNS(x, k_max, t_max);
1   répéter
2       k ← 1;
3       répéter
4           x' ← Shake(x, k)              /* Perturbation */;
5           x'' ← RechercheLocale(x') ;
6           NeighborhoodChange(x, x'', k) ;
        jusqu'à k = k_max;
7       t ← CpuTime() ;
    jusqu'à  t > t_max;
```

Figure 4. Algorithme VNS

La ligne numéro 5 de l'algorithme VNS peut être n'importe quelle procédure de recherche locale. Dans la version de base de la VNS, la procédure de recherche locale peut être notamment la procédure *FirstImprovement()* (Figure 2) ou encore *BestImprovement()* (Figure 1).

Il existe plusieurs variantes de la VNS. Nous nous intéressons à celle appelée GVNS (pour *General Variable Neighbourhhod Search*). La GVNS est la variante déduite de la VNS lorsque la procédure de recherche locale (i.e., ligne 5 de l'algorithme VNS) est remplacée par une procédure VND (pour *Variable Neighborhood Descent*). Cette dernière procédure permet de déterminer un minimum local en explorant un ensemble ordonné de v_{max} voisinages V_v (v est l'indice de chaque voisinage). L'algorithme VND est présenté à la Figure 5 et l'algorithme GVNS est présenté à la Figure 6.

Procédure VND(x, v_{max});

1 **répéter**

2 $v \leftarrow 1$;

3 **répéter**

4 $x' \leftarrow \arg min_{y \in V_v(x)} f(x)$ /* Trouver le meilleur voisin dans $V_v(x)$ */;

5 *NeighborhoodChange*(x, x', v) ;

jusqu'à $v = v_{max}$;

jusqu'à $pas\ d'amélioration\ possible$;

Figure 5. Algorithme VND

Nous rappelons que, dans l'algorithme VND, v_{max} est le nombre total des voisinages V_v considérés.

Procédure GVNS$(x, v_{max}, k_{max}, t_{max})$;

1 **répéter**

2 $k \leftarrow 1$;

3 **répéter**

4 $x' \leftarrow Shake(x, k)$;

5 $x'' \leftarrow VND(x', v_{max})$;

6 *NeighborhoodChange*(x, x'', k);

Jusqu'à $k = k_{max}$;

7 $t \leftarrow$ CpuTime() ;

Jusqu'à $t > t_{max}$;

Figure 6. Algorithme GVNS

Pour l'algorithme GVNS, nous notons que v_{max} est le nombre de voisinages V_v considérés dans la procédure VND, c'est-à-dire que la solution retournée par la procédure VND est un minimum local par rapport aux v_{max} voisinages V_v, alors que k_{max} est un paramètre à choisir. Ce dernier paramètre peut être vu comme une limite sur le degré de perturbation, via la fonction *Shake()*, à effectuer quand il n'y a pas d'amélioration d'une solution 'actuelle' x.

Nous avons présenté des généralités sur la recherche à voisinage variable, ainsi que sa variante GVNS. Nous remarquons que le cadre général de cette métaheuristique est simple et que les facteurs qui peuvent jouer le rôle le plus important sont : les voisinages à choisir ; l'ordre de ces voisinages ; et les procédures pour déterminer le meilleur voisin d'une certaine solution (voir la ligne 4 de l'algorithme VND). Nous présenterons dans la section suivante quelques voisinages pour le problème du voyageur de commerce avec fenêtres de temps et nous indiquerons ceux que nous avons considérés dans notre méthode. Quant à l'ordre choisi pour l'exploration des voisinages, ainsi que les procédures de détermination du meilleur voisin d'une solution, ils seront présentés dans le chapitre 4.

3.2 Quelques voisinages pour le TSPTW

Nous allons présenter quelques voisinages pour le TSPTW. Plusieurs articles discutent des mouvements et voisinages pour le TSPTW, par exemple Lin (1965), Lin et Kernighan (1973) et Funke, Grünert et Irnich (2005).

Les mouvements de type k-opt pour le TSPTW sont des mouvements qui consistent à supprimer k arcs d'une tournée et de les remplacer par k autres arcs de façon à ce que la solution résultante soit une tournée. Ces mouvements sont très connus, car ils généralisent, comme nous allons le voir, plusieurs mouvements connus sous d'autres noms.

L'un des voisinages que nous utilisons de notre méthode est le voisinage correspondant au mouvement 2-opt. Dans ce mouvement, deux arcs sont supprimés d'une tournée et sont remplacés par deux autres arcs de telle sorte à avoir une autre tournée.

Des mouvements qui consistent à bouger une chaîne de m clients consécutifs sont également très utilisés. En général, la valeur du nombre m ne dépasse pas trois. Nous appellerons

ces mouvements Or-opt-m. Dans ce mouvement, une chaîne de m clients est bougée un certain nombre de positions en avant ou en arrière. Nous avons considéré dans notre méthode les mouvements Or-opt-1 et Or-opt-2, i.e., bouger un client en avant ou en arrière et bouger une chaîne de deux clients en avant ou en arrière. Ces mouvements peuvent être vus comme des cas particuliers du mouvement 3-opt.

Nous considérons également dans notre méthode un voisinage appelé 1-opt (pour les noms des mouvements, nous utilisons les mêmes que ceux de Mladenovic et al. (2012)) qui correspond au mouvement qui consiste à échanger l'ordre de deux clients successifs. Ce mouvement peut être vu comme un cas particulier des mouvements Or-opt-1 et 2-opt.

Après avoir présenté le cadre général de la VNS et sa variante GVNS, et présenté les mouvements et voisinages que nous avons utilisés dans notre méthode, tous les ingrédients sont réunis pour pouvoir présenter en détail la méthode utilisée.

Chapitre 4 : Méthode proposée

Tel que cité dans l'introduction, nous adaptons la méthode de résolution du TSPTW-T présentée par Mladenovic et al. (2012) en introduisant quelques modifications permettant de résoudre efficacement le TSPTW-C, notamment au niveau de l'évaluation de la fonction objectif et de l'exploration des voisinages.

López-Ibáñez et al. (2013) indiquent qu'il existe des problèmes qui ont des structures semblables, mais qu'une variante a été plus étudiée qu'une autre, et qu'il est intéressant d'essayer d'adapter les méthodes qui ont donné de bons résultats pour la variante bien étudiée pour qu'elles puissent traiter de manière efficace la variante moins étudiée. Ceci a bien fonctionné pour leur méthode (voir la section 2.2.2), notamment pour l'adaptation de *Beam-ACO* qui est considérée comme l'état de l'art pour le TSPTW-C actuellement.

L'algorithme général de notre méthode est présenté comme suit -suivant les mêmes algorithmes présentés à la section 3.1 :

Procédure GVNS-TSPTW-C(x, k_{max}, t_{max});
1 $x \leftarrow PhaseConstructive()$;
2 **répéter**
3 $k \leftarrow 1$;
4 **répéter**
5 $x' \leftarrow Shake(x, k)$;
6 $x'' \leftarrow VND(x', 6)$;
7 *NeighborhoodChange*(x, x'', k);
Jusqu'à $k = k_{max}$;
8 $t \leftarrow$ CpuTime() ;
Jusqu'à $t > t_{max}$;

Figure 7. Algorithme GVNS-TSPTW-C

Notre méthode se déroule en deux phases. La première est appelée phase constructive et a pour objectif de trouver une solution réalisable au TSPTW-C à l'aide d'une heuristique VNS (la ligne 1 de l'algorithme GVNS-TSPTW-C). La procédure de recherche locale utilisée dans cette phase est de type *first improvement*. La deuxième phase, dite d'amélioration, se charge de l'amélioration de la solution réalisable trouvée à la phase constructive. Les procédures de recherche locale utilisées dans cette phase sont de type *best improvement* (voir la section 3.1 pour les algorithmes des procédures de recherche locale de type *first improvement* et *best improvement*). Dans la phase d'amélioration, notre méthode considère uniquement les solutions réalisables. Notons que le nombre de voisinages utilisés a été fixé à six (ligne 6 de la Figure 7). Ces voisinages correspondent aux mouvements: Or-opt-2 en avant, Or-opt-2 en arrière, 1-opt, Or-opt-1 en arrière, Or-opt-1 en avant et 2-opt (voir la section 3.2 pour les noms des mouvements). k_{max} est un paramètre à choisir qui représente une limite sur le degré de perturbation à effectuer. Une perturbation (ligne 5 de la Figure 7) correspond à effectuer k mouvements réalisables Or-opt-1.

4.1 Phase constructive

Savelsbergh (1985) prouve que le problème de trouver une solution réalisable pour le TSPTW est un problème fortement NP-complet, même lorsque la matrice des temps est symétrique et qu'elle satisfait l'inégalité triangulaire. Pour construire une solution réalisable, nous utilisons la méthode présentée dans Da Silva et Urrutia (2010).

Lorsque l'on traite le TSPTW avec des heuristiques, on peut représenter une solution comme un arrangement de tous les clients. On peut également inclure le dépôt origine au début de la liste et le dépôt destination à la fin de la liste. L'ordre des clients dans la liste correspond donc à l'ordre de visite des clients. Une telle représentation permet d'assurer que la solution est une tournée qui commence au dépôt origine, visite tous les n clients exactement une fois et revient au dépôt destination (qui est en réalité le même dépôt origine). Ainsi, les seules contraintes que l'on doit satisfaire, pour trouver une solution réalisable, sont celles relatives aux fenêtres de temps. Da Silva et Urrutia (2010) proposent donc un algorithme VNS qui vise à minimiser la fonction objectif définie par $f(x) = \sum_{i=1}^{n+1} \max\{0, A_i - b_i\}$ (voir la section 2.1.2 pour les notations). Cette équation représente en fait la somme des violations des contraintes

relatives aux fenêtres de temps. Ainsi, trouver une solution réalisable pour le TSPTW revient à trouver une solution pour laquelle la valeur de la fonction *f* est nulle.

L'algorithme de la phase constructive est le suivant :

```
    Procédure PhaseConstructive();
1   répéter
2     niveau ← 1;
3     x ← SolutionAleatoire();
4     x ← OrOpt1ConstructiveRechercheLocale(x);
5       tant que x n'est pas réalisable et niveau < niveauMax faire
6         x' ← Shake(x, niveau);
7         x' ← OrOpt1ConstructiveRechercheLocale(x');
8         NeighborhoodChange(x, x', niveau);
9       fin
10    t ← CpuTime() ;
    Jusqu'à x est réalisable ou t > t_max;
```

Figure 8. Algorithme PhaseConstructive

Pour chaque instance du TSPTW-C, la procédure *SolutionAleatoire()* génère une solution aléatoire *x* (probablement non réalisable, i.e., $f(x) > 0$). La procédure de perturbation *Shake(x,niveau)* réalise *niveau* mouvements Or-opt-1 aléatoirement (i.e., le client est choisi aléatoirement et sa nouvelle position est aussi choisie aléatoirement). Le paramètre *niveauMax* qui définit une limite sur le degré de la perturbation maximale a été choisi égal à 8 (*niveauMax=8*) (nous avons utilisé la même valeur utilisée dans Da Silva et Urrutia (2010)).

La procédure *OrOpt1ConstructiveRechercheLocale()* est la plus importante procédure dans la phase constructive. C'est une procédure de recherche locale de type *first improvement* (voir la section 3.1). Le seul mouvement utilisé dans cette procédure est le mouvement Or-opt-1 qui permet de bouger un client à une position donnée. La particularité de cette procédure est

qu'elle partitionne l'ensemble des clients en deux : l'ensemble des clients visités après la fin de leurs fenêtres horaires, que nous appellerons les clients non réalisables (nous rappelons que la solution identifiée grâce à la procédure *SolutionAleatoire()* n'est pas forcément réalisable) ; et l'ensemble des clients visités durant leurs fenêtres de temps, que nous appellerons les clients réalisables (nous rappelons que si on arrive à un client avant le début de sa fenêtre de temps, on peut attendre jusqu'à l'ouverture de sa fenêtre horaire pour le visiter). Le mouvement Or-opt-1 peut être également réparti en deux types : bouger un client en avant et bouger un client en arrière. Ainsi, on peut déduire quatre types de mouvements : bouger un client non réalisable en avant ; bouger un client non réalisable en arrière ; bouger un client réalisable en avant ; et bouger un client réalisable en arrière. Da Silva et Urrutia (2010) proposent donc que la procédure *OrOpt1ConstructiveRechercheLocale()* commence par explorer les mouvements qui ont plus de probabilité à mener à des solutions réalisables. L'ordre choisi est le suivant : bouger un client non réalisable en arrière ; bouger un client réalisable en avant ; bouger un client réalisable en arrière ; et bouger un client non réalisable en avant. Les auteurs présentent même une comparaison montrant que le fait d'ordonner les mouvements de cette façon conduit à trouver une solution réalisable plus rapidement par rapport à une implémentation où la procédure *OrOpt1ConstructiveRechercheLocale()* exécute les mouvements Or-opt-1 pour tous les clients sans un partitionnement préalable des mouvements.

Les résultats obtenus par cette phase de construction ont été bons. Nous jugeons que nous arrivons à trouver des solutions réalisables rapidement. Nous avons suivi les grandes lignes de la phase constructive présentée par Da Silva et Urrutia (2010), mais nous ne pouvons pas prétendre avoir programmé dans les moindres détails la même phase constructive.

Après avoir présenté la phase constructive permettant de trouver une solution réalisable, nous présenterons par la suite la phase d'amélioration où nous essayons d'améliorer la solution réalisable identifiée.

4.2 Phase d'amélioration

Nous allons rappeler d'abord la méthode que nous avons adapté en citant notamment les principaux changements que nous avons introduits.

4.2.1 La méthode à adapter pour la phase d'amélioration

Mladenovic et al. (2012) résolvent le TSPTW-T avec la métaheuristique GVNS. Leur méthode est composée de deux étapes. La première étape utilise l'algorithme de la phase constructive présenté dans la section 4.1. Lors de la deuxième étape, leur algorithme essaie, avec une heuristique GVNS, d'améliorer la solution réalisable trouvée à la première étape. Les voisinages (on utilise le mot voisinage pour désigner le voisinage correspondant à un mouvement donné) utilisés pour la VND sont, dans l'ordre : 1-opt, Or-opt-2 en arrière, Or-opt-2 en avant, Or-opt-1 en arrière, Or-opt-1 en avant et 2-opt. Les auteurs ont utilisé notamment un vecteur que nous appelons dans ce document *écart*. Ce vecteur indique pour chaque nœud *i* de la tournée (y compris le dépôt origine et le dépôt destination) de combien on peut retarder l'arrivée au nœud *i* de sorte que la partie de la tournée (le chemin) qui commence au nœud *i* et se termine au dépôt destination reste réalisable. Ce vecteur est défini de façon récurrente comme suit (voir la section 2.1.2 pour les notations utilisées)

$$e_i = min\{e_{i+1} + max\{0, a_i - A_i\}, b_i - A_i\}\,, \quad i = 0 \dots n$$

avec $e_{n+1} = b_{n+1} - A_{n+1}$, et *i* représente le client qui se trouve à la position *i* de la tournée ($i = 0$ représente le dépôt origine et $i = n + 1$ représente le dépôt destination).

e_i est appelé l'écart au nœud *i*.

Dans toute la suite, nous considérons qu'une tournée est représentée par une suite ordonnée *(0,1,2,...,i,...,n,n+1)* où *i* représente le client qui se trouve à la position *i*. Nous allons toutefois utiliser uniquement *i* pour référer ce client (pour ne pas alourdir le texte en écrivant à chaque fois le client qui se trouve à la position *i*), la position zéro est occupée par le dépôt origine (qui est également appelé nœud *0* ou client *0*) et la position *n+1* est occupée par le dépôt destination (appelé également nœud *n+1* ou client *n+1*).

Étant donné une solution réalisable, le vecteur *écart* est important pour évaluer la réalisabilité d'un voisin (c'est-à-dire une solution résultant après avoir effectué un mouvement parmi les mouvements cités auparavant). En effet, pour un mouvement appliqué à cette solution réalisable, la structure de la solution résultante a, dans la plupart des cas, la structure suivante :

Figure 9. Structure d'une solution après un mouvement

La partie contenant les nœuds entre *0* et *p-1* (inclus) reste inchangée : les positions sont occupées par les mêmes clients et les temps d'arrivée ne changent pas. La partie contenant les nœuds allant de la position *p* jusqu'à *q* est modifiée éventuellement : les positions sont probablement occupées par d'autres clients et les nouveaux temps d'arrivée sont différents éventuellement. La dernière partie contenant les nœuds allant de la position *c** jusqu'au dépôt destination *n+1* est éventuellement modifiée en ce qui concerne les temps d'arrivée aux clients, mais reste inchangée vis-à-vis des clients qui occupent les positions allant de *c** à *n+1*. La position *c** (comme indiqué auparavant, nous allons utiliser *c** pour désigner le client qui se trouve à la position *c**) peut être vue donc comme la plus petite position *j* à partir de laquelle le chemin *(j,j+1,...,n+1)* est inchangé vis-à-vis des clients qui occupent les positions de *j* à *n+1* (les temps d'arrivée à ces clients sont modifiés éventuellement). Donc, étant donné une solution réalisable, et après avoir appliqué un mouvement à cette solution, pour vérifier la réalisabilité de la solution résultante, il suffit de :

- vérifier la réalisabilité des clients entre *p* et *q*; si au moins un client n'est pas réalisable alors la solution résultante n'est pas réalisable et on arrête le processus de vérification de la réalisabilité; si les clients sont tous réalisables alors on passe à la prochaine étape;
- mettre à jour le temps d'arrivée au client *c** (l'ancien temps d'arrivée est noté A_{c*}et le nouveau temps d'arrivée est nommé A_{c*}^{N}); si ($A_{c*}^{N} - A_{c*} > e_{c*}$) alors la solution résultante n'est pas réalisable; sinon (i.e., $A_{c*}^{N} - A_{c*} \leq e_{c*}$) alors la solution résultante est réalisable.

Ainsi, le vecteur *écart* permet de vérifier la réalisabilité d'une solution sans avoir à parcourir la tournée (de la position *p* jusqu'à la position *n+1*, voir Figure 9) et mettre à jour les temps d'arrivée de tous les clients en comparant ces temps d'arrivée aux fenêtres de temps de correspondantes.

Nous notons que le vecteur *écart* est également important pour éliminer des temps d'attente inutiles. Pour ce faire, on doit retarder le temps du départ du dépôt origine. Si on retarde le temps du départ du dépôt d'une quantité supérieure à $\sum_{i=0}^{n+1} w_i$ (la quantité $\sum_{i=0}^{n+1} w_i$ représente le temps d'attente total dans une tournée), alors le temps d'arrivée au dépôt destination sera retardé également, ce qui n'est pas intéressant dans le cas du TSPTW-C. Si on retarde le temps du départ du dépôt d'une quantité égale à $\sum_{i=0}^{n+1} w_i$, alors le temps d'arrivée au dépôt destination ne changera pas et les temps d'attente seront éliminés, mais on doit s'assurer de la réalisabilité de la solution résultante. Pour que la nouvelle solution (après avoir retardé le temps du départ du dépôt) soit réalisable, le retard doit être inférieur ou égal à la valeur e_0. Ainsi, si le temps du départ du dépôt origine D_0 est remplacé par la quantité $D_0 + min\{\sum_{i=0}^{n+1} w_i ; e_0\}$ (où w_i représente le temps d'attente au client *i*, et e_0 est la valeur de l'écart au dépôt origine), alors on éliminera une partie ou la totalité des temps d'attente tout en assurant que le temps d'arrivée au dépôt destination reste inchangé et que la solution demeure réalisable.

Ayant en main un outil fort de vérification de la réalisabilité d'un mouvement, le seul souci est de pouvoir vérifier de manière efficiente la rentabilité d'un mouvement (si le mouvement améliore la solution réalisable actuelle ou non). Pour le TSPTW-T (minimisation de la distance ou du temps effectif du trajet), la tâche est facile, car il suffit de comparer la somme des distances (des temps) des arcs supprimés à la somme des distances (temps) des nouveaux arcs pour déterminer et quantifier la rentabilité d'un mouvement.

Pour le cas du mouvement 2-opt (parmi les mouvements considérés, seul le mouvement 2-opt implique qu'une partie de la tournée soit inversée), si la matrice des distances n'est pas symétrique, alors il faut faire attention aux arcs inversés lors de la vérification de la rentabilité d'un mouvement même pour la variante TSPPTW-T.

Nous présentons maintenant les principales remarques, positives et négatives, concernant la méthode présentée par Mladenovic et al. (2012), en indiquant leur relation avec la variante qui nous intéresse TSPTW-C (minimisation du temps d'arrivée au dépôt destination):

- Le vecteur *écart* permet une vérification efficiente de la réalisabilité d'un mouvement pour le TPSTW-T et il en est de même pour le TSPTW-C.
- La vérification de la rentabilité d'un mouvement est relativement facile et rapide pour le TSPTW-T. En ce qui concerne le TSPTW-C, la vérification de la rentabilité d'un mouvement n'est pas évidente. En principe, il faut parcourir toute la tournée (à partir de la position p jusqu'à $n+1$, voir Figure 9). Ainsi, la vérification de la rentabilité d'un mouvement devient gourmande en temps de calcul pour le TSPTW-C. Pour remédier à ce problème, nous présenterons une méthode efficiente pour la vérification de la rentabilité d'un mouvement à la section 4.2.2.3.
- La méthode proposée par Mladenovic et al. (2012) pour le TSPTW-T a donné de bons résultats. Ainsi, nous utiliserons les mêmes voisinages considérés (cependant, des résultats expérimentaux nous ont conduit à modifier l'ordre des voisinages dans la VND pour obtenir de bons résultats pour le TSPTW-C).
- La méthode proposée par Mladenovic et al. (2012) n'utilise aucun type de prétraitement, notamment pour l'élimination des arcs incompatibles (des arcs conduisant à des solutions non réalisables). Nous allons intégrer des tests de prétraitement de base permettant d'éliminer des arcs incompatibles. En identifiant ces arcs incompatibles et en explorant chaque voisinage dans un ordre déterminé, on peut réduire le nombre de solutions à visiter tout en étant sûr de visiter toutes les solutions réalisables (nous notons que la façon utilisée par Mladenovic et al. (2012), pour parcourir un voisinage, ne garantit pas que toutes les solutions réalisables soient visitées, et donc ne garantit pas que les procédures de recherche locale retournent une solution qui est un minimum local).

4.2.2 La méthode proposée pour la phase d'amélioration

L'algorithme de la méthode proposée pour la phase d'amélioration est celui présenté à la Figure 7 (sans tenir compte de la ligne 1 de la dite figure). Il est à noter que, pour la phase d'amélioration, la procédure *Shake(x,k)* effectue *k* mouvements Or-opt-1 réalisables (dans la phase constructive on n'impose pas que les mouvements effectués par la procédure *Shake(x,k)* soient réalisables).

4.2.2.1 Un prétraitement de base

Nous effectuons une étape de prétraitement de base qui permet d'identifier des arcs incompatibles, i.e., des arcs qui ne peuvent jamais exister dans une solution réalisable. Un arc *(i,j)* est incompatible si $a_i + t_{ij} > b_j$ (voir la section 2.1.2 pour les notations), c'est-à-dire qu'un arc *(i,j)* est incompatible si on ne peut pas arriver au client *j* avant la fin de sa fenêtre et ce même si on quitte le client *i* le plus tôt possible. En marquant ces arcs incompatibles, l'algorithme évite de vérifier des solutions qui sont clairement non réalisables. De plus, et puisque les temps t_{ij} sont positifs et satisfont l'inégalité triangulaire, nous pouvons même déduire que si un arc *(i,j)* est incompatible, alors toute solution dans laquelle le client *j* est visité après (pas forcément exactement après) le client *i* est une solution non réalisable. Nous verrons plus tard que l'ordre de l'exploration d'un voisinage est crucial pour bien exploiter l'information déduite par les arcs incompatibles.

4.2.2.2 L'ordre des voisinages

Les voisinages que nous avons considérés dans la VND sont, dans l'ordre : Or-opt-2 en avant, Or-opt-2 en arrière, 1-opt, Or-opt-1 en arrière, Or-opt-1 en avant et 2-opt (nous appelons cet ordre VND_Ordre_1 dans le Tableau I). Nous avons testé quelques ordres différents de ces voisinages, notamment l'ordre proposé dans Mladenovic et al. (2012), à savoir 1-opt, Or-opt-2 en arrière, Or-opt-2 en avant, Or-opt-1 en arrière, Or-opt-1 en avant et 2-opt (nous appelons cet ordre VND_Ordre_2 dans le Tableau I). Nous avons testé notre méthode sur quatre groupes d'instances (nommés dans ce document AFG, DUMAS, GENDREAU et OHLMANN; voir la section 5.1.1 pour la description et la source des instances). Le Tableau I présente les valeurs correspondant à la meilleure solution trouvée, pour le TSPTW-C, pour les ordres VND_Ordre_1

et VND_Ordre_2. Les instances rapportées dans le Tableau I sont celles pour lesquelles nous avons constaté des différences, entre VND_Ordre_1 et VND_Ordre_2, dans la valeur de la meilleure solution trouvée après 15 exécutions.

Instance	VND_Ordre_1	VND_Ordre_2
rbg172a	17 783	17 784
n40w120.002	557	565
n60w120.003	541	546
n80w120.004	644	645
n80w160.002	654	665
n80w160.005	645	647
n150w160.002	890	891
n200w120.001	1089	1091
n200w140.004	1100	1102
n150w120.003	910	909

Tableau I. Influence de l'ordre des voisinages sur les résultats

Nous remarquons que, pour la même limite de temps, l'ordre VND_Ordre_2 n'arrive pas à trouver la meilleure solution trouvée par l'ordre VND_Ordre_1 pour neuf instances, alors que l'ordre VND_Ordre_1 n'arrive pas à trouver la meilleure solution trouvée par l'ordre VND_Ordre_2 pour une seule instance (n150w120.003). Le Tableau I montre que l'ordre des voisinages dans la VND a une influence sur la valeur de la meilleure solution trouvée.

En analysant l'algorithme de la VND (voir Figure 5), nous déduisons une remarque importante concernant l'ordre des voisinages à considérer. En effet, si on utilise des procédures de recherche locale permettant de retourner un minimum local -ce qui est le cas généralement, alors il est inutile de considérer un voisinage V_i après un voisinage V_j si le premier voisinage est un cas particulier du second. Dans notre cas, il est inutile de considérer le voisinage 1-opt après le voisinage Or-opt-1 (en avant ou en arrière) ou le voisinage 2-opt, car le voisinage 1-opt est un cas particulier de ces voisinages (voir la section 3.2).

4.2.2.3 La vérification de la rentabilité

Nous avons vu dans la section 4.2.1 que le vecteur *écart* offre un outil efficient pour la vérification de la réalisabilité d'un mouvement. En effet, il suffit de connaitre le nouveau temps d'arrivée au client $c*$ (nous rappelons que le client $c*$ désigne en fait le client qui se trouve à la plus petite position j à partir de laquelle le chemin $(j,j+1,\ldots,n+1)$ est inchangé vis-à-vis des clients qui occupent les positions de j à $n+1$, voir la Figure 9 et la section 4.2.1 pour plus d'explications). Pour pouvoir bien profiter de cette approche concernant la vérification de la réalisabilité, nous présentons une façon permettant la vérification de la rentabilité d'un mouvement de façon efficiente.

Pour ce faire, nous définissons un vecteur, appelé *gain maximal*, qui va nous servir par la suite pour vérifier la rentabilité d'un mouvement. Ce vecteur est calculé au même moment que le vecteur *écart* avec la relation récurrente suivante :

$$gainMax_i = min\{gainMax_{i+1}, A_i - a_i\}\,, \quad i = 0 \ldots n$$

avec $gainMax_{n+1} = A_{n+1} - a_{n+1}$, a_i est le temps de début de la fenêtre horaire du client i, A_i est le temps d'arrivée au client i, et i représente le client qui se trouve à la position i de la tournée ($i = 0$ représente le dépôt origine et $i = n + 1$ représente le dépôt destination).

Le vecteur *gain maximal* peut aussi être exprimé, pour chaque nœud i, comme $gainMax_i = \min_{j\in\{i,i+1,\ldots,n+1\}}\{A_j - a_j\}$. Nous expliquerons par la suite comment ce vecteur peut être exploité pour vérifier la rentabilité d'un mouvement.

Dans le cas du TSPTW-C, étant donné une solution x, et en appliquant un mouvement à cette solution, nous obtenons une solution résultante x^N. Le mouvement appliqué est rentable (nous le définissons ainsi, il peut y avoir d'autres définitions de la rentabilité) si $A_{n+1}(x^N) < A_{n+1}(x)$ où $A_{n+1}(y)$ désigne le temps d'arrivée au dépôt d'une solution y.

Pour la solution x, le temps d'arrivée au client $c*$ est A_{c*}. En supposant que nous connaissons le nouveau temps d'arrivée au client $c*$, i.e., A^N_{c*}, nous remarquons que :

- Si $A^N_{c*} \geq A_{c*}$ alors $A_{n+1}(x^N) \geq A_{n+1}(x)$.

- Si $A^N_{c*} < A_{c*}$ et $gainMax_{c*} \leq 0$ alors $A_{n+1}(x^N) = A_{n+1}(x)$. En effet, tout le gain au niveau du temps d'arrivée au client *c** est "absorbé" par le premier (par rapport à l'ordre de visite qui commence à *c** et finit à *n+1*) client *k* pour lequel $A_k \leq a_k$.
- Si $A^N_{c*} < A_{c*}$ et $gainMax_{c*} > 0$ alors $A_{n+1}(x^N) < A_{n+1}(x)$, et on a $A_{n+1}(x^N) = A_{n+1}(x) - min\{gainMax_{c*}, A_{c*} - A^N_{c*}\}$. En effet, tout le gain au niveau du client *c** est propagé jusqu'au dépôt *n+1* si $gainMax_{c*} \geq A_{c*} - A^N_{c*}$, cependant si $gainMax_{c*} < A_{c*} - A^N_{c*}$ alors une partie du gain est "absorbée".

Nous avons présenté donc une façon permettant de vérifier la rentabilité d'un mouvement en tenant compte du temps d'arrivée au client *c**. De plus, dans le cas de la rentabilité d'un mouvement, nous pouvons prévoir exactement le temps d'arrivée au dépôt *n+1* sans avoir à parcourir le chemin *(c*,...,n+1)*. Nous notons qu'une approche qui ressemble à la nôtre a été utilisée par Savelsbergh (1992) pour évaluer la rentabilité d'un mouvement dans le cas du TSPTW-D (minimisation de la durée de la tournée). Dans cette approche, l'auteur présente des théorèmes de concaténation et des méthodes qui permettent de définir des quantités, et les maintenir à jour au cours de chaque mouvement, pour pouvoir essentiellement garder l'information sur le temps d'attente total de chaque chemin. Ces temps d'attente sont utilisés pour déterminer si un mouvement est rentable et quantifier l'impact d'un mouvement sur la valeur objectif dépendamment de la somme des temps d'attente des chemins.

Nous notons également que la méthode que nous avons présentée pour la vérification de la rentabilité d'un mouvement ne dépend pas du mouvement lui-même. En effet, pour n'importe quel mouvement, il suffit de déterminer le client *c** (i.e., le client qui se trouve à la plus petite position *j* à partir de laquelle le chemin *(j,...,n+1)* est inchangé vis-à-vis des clients qui occupent les positions de *j* à *n+1*), de calculer le nouveau temps d'arrivée au client *c**, puis de déterminer la rentabilité du mouvement sans avoir à parcourir le chemin *(c*,...,n+1)*. Nous rappelons que le vecteur *écart* (présenté à la section 4.2.1) permet également de vérifier la réalisabilité du chemin *(c*,...,n+1)* sans avoir à le parcourir.

4.2.2.4 L'exploration des voisinages

L'ordre dans lequel on explore un voisinage peut être d'une importance cruciale (quant au nombre des voisins évalués et au temps nécessaire pour explorer tout le voisinage). Nous explorons chaque voisinage dans un ordre lexicographique (voir Savelsbergh (1992)).

Prenons comme exemple le voisinage correspondant au mouvement Or-opt-1 en avant (le mouvement qui consiste à bouger un client à une position supérieure à la position actuelle du client). Une procédure de recherche locale (PRL) de type *best improvement* (voir la section 3.1) a pour objectif de trouver un minimum local pour ledit voisinage. Étant donné une solution actuelle réalisable *x*, la PRL doit évaluer tous les voisins (un voisin de *x* est une solution obtenue après avoir effectué un mouvement Or-opt-1) de la solution actuelle *x*. S'il y a un ou plusieurs voisins qui améliorent la solution *x* (et qui sont réalisables bien sûr), alors la PRL choisit la meilleure solution x^M -en fait, une parmi les solutions qui sont les meilleures à égalité- et la solution x^M devient la solution "actuelle". Ce processus est répété jusqu'à ce qu'il n'y ait plus d'amélioration possible (i.e., un minimum local a été identifié). Pour le voisinage considéré, la PRL doit, pour chaque client, tester tous les mouvements Or-opt-1 en avant. Pour ce faire, plusieurs ordres sont possibles. Prenons l'exemple des deux ordres suivants : *OrOpt1Ordre1* (Figure 10) est l'ordre lexicographique dans lequel le client à la position *p* est bougé à la position *p+1*, puis à la position *p+2*, et ainsi de suite jusqu'à la position *n* ; et *OrOpt1Ordre2* (Figure 11) est l'ordre dans lequel le client à la position *p* est bougé d'abord à la position *n*, puis à la position *n-1*, et ainsi de suite jusqu'à la position *p+1*.

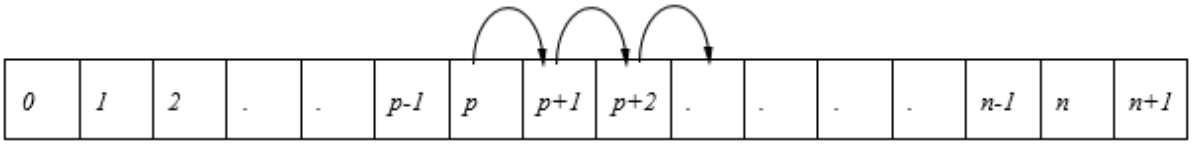

Figure 10. Ordre *OrOpt1Ordre1* pour Or-opt-1 en avant

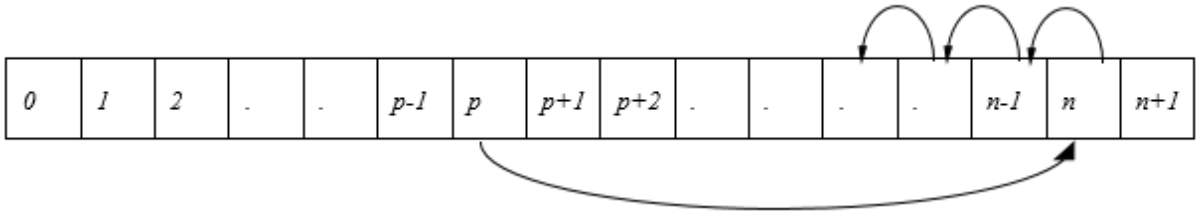

Figure 11. Ordre *OrOpt1Ordre2* pour Or-opt-1 en avant

Pour faciliter la compréhension de cette partie, nous n'allons pas confondre client et position. Nous utiliserons donc la notation C_i pour désigner le client i, et la notation CP_i pour désigner le client qui se trouve à la position i.

Pour l'ordre *OrOpt1Ordre1*, on vise à évaluer tous les mouvements Or-opt-1 en avant pour un client C_i qui se trouve initialement à la position p (voir la Figure 12 pour l'algorithme que nous utilisons). Dans la ligne 1 de l'algorithme, j représente la position actuelle de C_i. A la ligne 2, nous identifions le client *SuccesseurCi* qui est actuellement le successeur du client C_i. A la ligne 3, nous identifions le client qui se trouve à la position c^* (nous rappelons que c^* est la plus petite position k à partir de laquelle le chemin $(k,...,n+1)$ est inchangé vis-à-vis des clients qui occupent les positions de k à $n+1$, voir les sections 4.2.1 et 4.2.2.3 pour plus de détails concernant la position c^* et son importance dans notre méthode). À la ligne 4, nous effectuons un test concernant les arcs incompatibles: si l'arc $(SuccesseurCi,C_i)$ est incompatible, alors le client C_i ne peut pas être dans une position supérieure à la position du client *SuccesseurCi* dans n'importe quelle solution réalisable. Dans ce cas, nous arrêtons l'algorithme sans avoir à évaluer toutes les prochaines positions et nous passons donc à évaluer les mouvements pour un autre client (rappelons que l'algorithme de la Figure 12 concerne les mouvements Or-opt-1 en avant pour un seul client). Si nous avions utilisé l'ordre *OrOpt1Ordre2*, nous n'aurions pas pu profiter de l'information concernant les arcs incompatibles pour éliminer tout un ensemble de solutions non réalisables. Si l'arc $(SuccesseurCi,C_i)$ est compatible, alors, à la ligne 7, on met à jour le nouveau temps d'arrivée au *successeurCi* (i.e., $A^N_{SuccesseurCi}$) à partir du client CP_{j-1} (cette mise à jour s'effectue en $O(1)$). Nous notons qu'il est inutile de vérifier la réalisabilité du client *SuccesseurCi*, car il était réalisable avant le mouvement et l'inégalité triangulaire assure qu'il ne peut pas être visité plus tard que dans la solution réalisable actuelle. À la ligne 8, on met à jour

(en *O(1)*) le nouveau temps d'arrivée à C_i (i.e., A_{Ci}^N) et on vérifie, à la ligne 9, s'il satisfait les contraintes sur les fenêtres de temps du dit client. Si $A_{Ci}^N > b_{Ci}$ (où b_{Ci} est le temps de fin de la fenêtre horaire du client *Ci*), alors on arrête l'algorithme (et on passe à un autre client), car l'inégalité triangulaire assure que la valeur de A_{Ci}^N ne peut pas diminuer pour les prochaines positions supérieures à la position *j*. Si le client *Ci* est réalisable, alors la ligne 12 se charge de la mise à jour du nouveau temps d'arrivée au client C_{c*} (i.e., $A_{C_{c*}}^N$) à partir du client *Ci*. Ceci est effectué en *O(1)* également. Avec l'information sur le nouveau temps d'arrivée au client *Cc** et en connaissant également l'ancien temps d'arrivée au client *Cc** (i.e., $A_{C_{c*}}$), nous pouvons vérifier (en *O(1)*) si la nouvelle solution est réalisable ou non à l'aide du vecteur *écart* (voir la section 4.2.1). Si la solution s'avère non réalisable, alors l'algorithme continue (ligne 1). Si la solution est réalisable, alors on vérifie (en *O(1)*) si le mouvement effectué est rentable grâce à l'information sur $A_{C_{c*}}^N$ et le vecteur *gainMax* (voir la section 4.2.2.3). Si le mouvement n'est pas rentable, alors on passe à l'évaluation de la prochaine position (ligne 1). Si le mouvement s'avère rentable, alors, avant de passer à la prochaine position (ligne 1), on calcule (en *O(1)*) le temps d'arrivée au dépôt destination A_{n+1} et on le compare au meilleur temps d'arrivée au dépôt actuellement connu par la procédure de recherche locale correspondant au mouvement Or-opt-1 en avant (l'algorithme de la Figure 12 est en fait une partie de l'algorithme d'une procédure de recherche locale de type *best improvement*).

```
1   pour j = p jusqu'à n − 1 faire  /* j est la position actuelle de Ci */
2       SuccesseurCi ← CP_{j+1} ;
3       C_{c*} ← CP_{j+2} ;
4       si l'arc (SuccesseurCi, C_i) est incompatible alors
5           arrêter ;
6       sinon
7           mettre à jour A^N_{SuccesseurCi} ; /* à partir de CP_{j−1} */
8           mettre à jour A^N_{Ci} ;  /* à partir de SuccesseurCi */
9           si Ci n'est pas réalisable alors
10              arrêter ;
11          sinon
12              mettre à jour A^N_{C_{c*}} ;  /* à partir de Ci */
13              si la tournée est réalisable alors /* à l'aide du vecteur écart */ ;
14                  vérifier la rentabilité du mouvement ; /* à l'aide de gainMax */
15              fin
16          fin
17      fin
18  fin
```

Figure 12. Algorithme pour bouger un client en avant

Pour l'ordre *OrOpt1Ordre2*, le principal inconvénient est le fait que nous ne pouvons pas profiter des lignes 5 et 10 de l'algorithme issu de l'ordre *OrOpt1Ordre1* ou d'un test similaire permettant d'éliminer toute une partie de l'espace de recherche.

Nous déduisons donc que les mouvements Or-opt-1 en avant sont évalués d'une façon performante.

Nous avons présenté le voisinage correspondant au mouvement Or-opt-1 en avant en donnant plusieurs détails. Nous allons présenter par la suite les grandes lignes des autres voisinages.

Les voisinages correspondant aux mouvements Or-opt-2 en avant et 1-opt (voir la section 3.2 pour plus de détails sur ces voisinages) sont évalués presque de la même façon que le voisinage correspondant au mouvement Or-opt-1 en avant.

Le voisinage correspondant au mouvement Or-opt-1 en arrière est exploré, pour un client C_i, qui est initialement à la position *p*, en bougeant ce client à la position *p-1*, puis à la position *p-2*, et ainsi de suite jusqu'à la position *1* (Figure 13). Nous remarquons donc que la position *c** (la plus petite position *k* à partir de laquelle le chemin *(k,k+1,...,n+1)* est inchangé vis-à-vis des clients qui occupent les positions de *k* à *n+1*) reste inchangée, en utilisant cet ordre, lorsque l'on considère le client C_i. En effet, la position c* est égale à la position *p+1* (Figure 13). En ce qui concerne l'utilisation de l'information offerte par les arcs incompatibles, là aussi on arrête de bouger le client *Ci* en arrière lorsqu'un arc incompatible "apparait" dans la solution résultante. Notons finalement que lorsque le client C_i est bougé, à une étape donnée, à la position *q*, la vérification de la réalisabilité et de la rentabilité de la solution résultante ne s'effectue pas en *O(1)*. Cette vérification nécessite : la mise à jour des temps d'arrivée des clients qui se trouvent dans les positions de *q* à *p+1* ; la vérification de la réalisabilité des clients qui occupent les positions de *q+1* à *p* (l'inégalité triangulaire assure la réalisabilité du client qui occupe la position *q* -i.e., le client C_i- car il était réalisable lorsqu'il occupait la position *p* et il a été bougé en arrière). La réalisabilité du client qui occupe la position *p+1* (i.e., la position *c**) serait vérifiée à l'aide du processus de la vérification de la réalisabilité du chemin *(c*,...,n+1)* décrit dans la section 4.2.1; et enfin l'application des processus de vérification de la réalisabilité (section 4.2.1) et de la rentabilité (section 4.2.2.3).

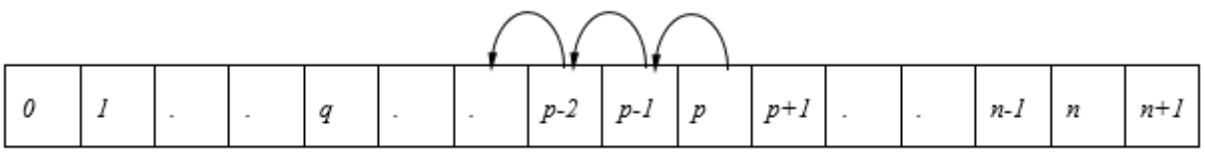

Figure 13. Ordre de parcours pour Or-opt-1 en arrière

Le raisonnement concernant le voisinage correspondant au mouvement Or-opt-2 en arrière est quasi identique au raisonnement que nous venons de présenter pour le voisinage correspondant au mouvement Or-opt-1 en arrière.

Pour le mouvement 2-opt, nous avons considéré dans notre travail le mouvement suivant:

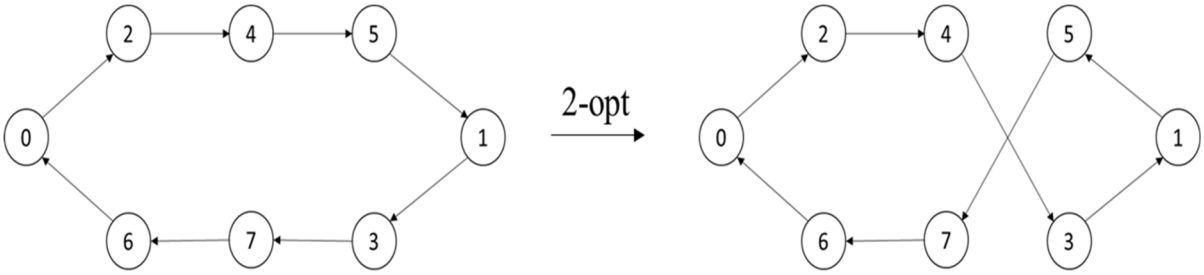

Figure 14. Mouvement 2-opt

Dans l'exemple de la Figure 14, nous avons choisi de représenter le dépôt uniquement avec le nœud *0* pour insister sur le fait que la solution résultante après un mouvement 2-opt doit être une tournée. Dans ladite figure, le mouvement 2-opt considéré consiste à supprimer les deux arcs *(4,5)* et *(3,7)*, les remplacer par les deux arcs *(4,3)* et *(5,7)* et inverser l'orientation des arcs *(5,1)* et *(1,3)*.

L'ordre que nous avons considéré pour parcourir le voisinage correspondant au mouvement 2-opt est l'ordre lexicographique expliqué par la Figure 15 (la source de cette figure est Savelsbergh (1992))

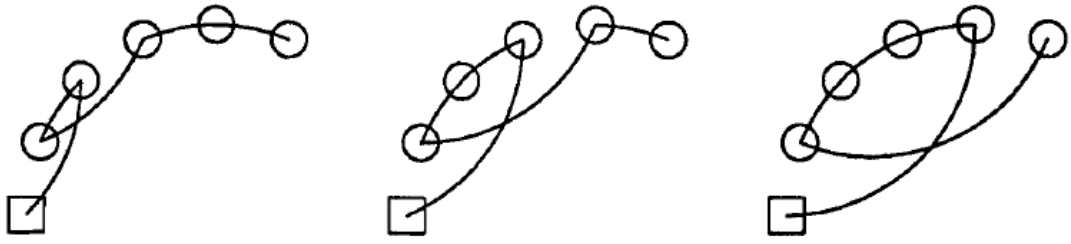

Figure 15. Ordre lexicographique pour 2-opt

Nous présentons à la Figure 16 la structure générale d'une solution après l'application du mouvement 2-opt. La notation CP_i désigne le client qui occupait la position i avant l'application du mouvement 2-opt.

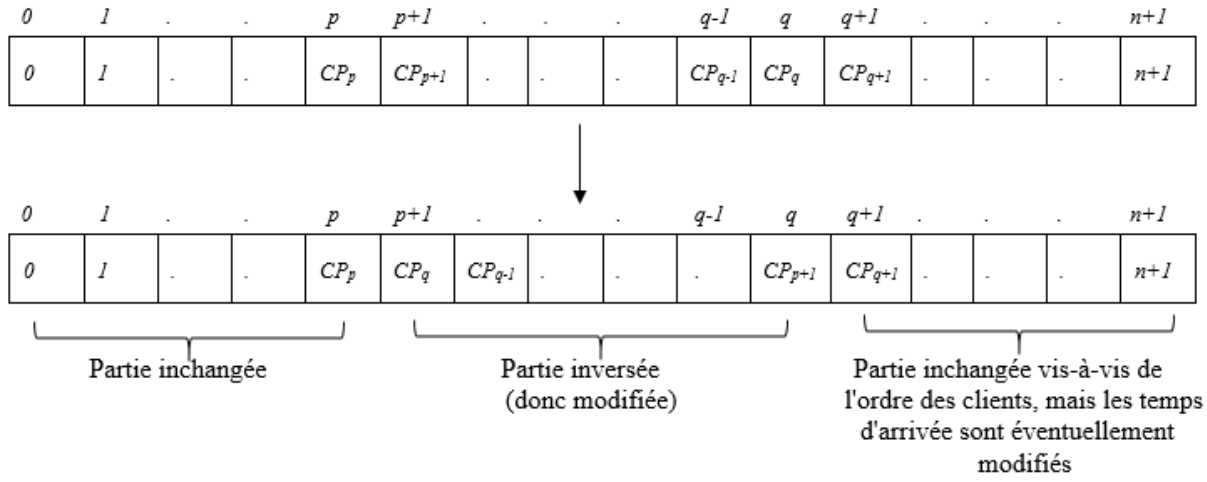

Figure 16. Structure d'une solution après le mouvement 2-opt

Le mouvement 2-opt présenté à la Figure 16 est celui qui supprime les arcs (CP_p,CP_{p+1}) et (CP_q,CP_{q+1}), les remplace par les arcs (CP_p, CP_q) et (CP_{p+1},CP_{q+1}) et inverse le chemin entre les clients CP_{p+1} et CP_q. En parcourant le voisinage correspondant au mouvement 2-opt avec l'ordre lexicographique (présenté à la Figure 15), nous profitons de quelques tests qui permettent d'éliminer un ensemble de solutions non réalisables sans avoir à les évaluer. En effet, l'ordre lexicographique peut être vu comme suit : pour chaque client CP_p, on veut parcourir les solutions déduites par l'application du mouvement 2-opt présenté à la Figure 16 où la position q varie de $p+2$ jusqu'à n. Ainsi, pour un client donné CP_p, après un mouvement 2-opt, on doit mettre à jour les temps d'arrivée des clients occupant les positions de $p+1$ jusqu'à $q+1$ (notons que la position $q+1$ correspond à chaque étape à la position c^*), puis vérifier la réalisabilité des clients occupant les positions de $p+2$ jusqu'à q (le client à la position $p+1$ est réalisable car la matrice des temps T satisfait l'inégalité triangulaire, et la vérification de la réalisabilité du client qui occupe la position $q+1$ (i.e., la position c^*) est incluse dans la procédure de vérification de la réalisabilité présentée en détails à la section 4.2.2). Si l'un de ces clients -les clients occupant

les position de *p+2* à *q*- n'est pas réalisable, alors on arrête le processus de parcours pour le client CP_p (et on passe au client suivant), car tous les prochains mouvements 2-opt, c'est-à-dire les prochaines positions *q*, conduiront à des solution non réalisables. Si tous les clients occupant les positions allant de *p+2* jusqu'à *q* sont réalisables, alors on déroule le processus de vérification de la réalisabilité (section 4.2.1) et de la rentabilité (section 4.2.2.3). Nous rappelons que la position *c** correspond à la position *q+1*.

Nous avons décrit les ordres de parcours des voisinages correspondant aux mouvements considérés par notre méthodes, à savoir : Or-opt-2 en avant, Or-opt-2 en arrière, 1-opt, Or-opt-1 en arrière, Or-opt-1 en avant et 2-opt. Nous avons expliqué les tests permettant d'éliminer des portions non réalisables de l'espace de recherche, et nous avons présenté les processus permettant de vérifier la réalisabilité et la rentabilité d'un mouvement sans avoir, dans la majorité des cas, à parcourir toute la solution résultante.

Les procédures de recherche locale (PRL) utilisées pour trouver le minimum local de chaque voisinage considéré sont de type *best improvement* (voir la Figure 1). Pour chaque PRL, nous notons que les composants des vecteurs *écart* (voir 4.2) et *gainMax* (voir 4.2.2.3) sont mis à jour uniquement lorsqu'il y a un changement de la solution "actuelle" (i.e., si $f(x) < f(x')$ dans l'algorithme de la Figure 1) considérée par la PRL.

Nous donnons également un détail concernant l'implémentation des PRL, car l'implémentation joue un rôle important lorsqu'il s'agit de procédures basées sur la recherche locale qui sont exécutées un grand nombre de fois : à chaque itération de la PRL (lignes 2 et 3 de la Figure 1), nous essayons de trouver le meilleur voisin à partir d'une solution actuelle *x*, donc nous parcourons les voisins pour les évaluer. Le parcours des voisins implique l'application d'un mouvement à une solution donnée, nous proposons cependant de ne pas appliquer un mouvement réellement, et de se limiter à simuler l'impact d'un mouvement sur une solution donné, ainsi la solution actuelle *x* est inchangée (physiquement) durant toute une itération de la PRL, et elle est changée uniquement si $f(x) < f(x')$ (voir la Figure 1) en appliquant une seule fois le mouvement qui a été jugé comme étant celui qui mène au meilleur voisin (nous éviterons ainsi d'effectuer des mouvements inutiles sur la structure physique de la solution).

Nous avons présenté dans ce chapitre la méthode que nous avons proposée pour résoudre le TSPTW-C (minimisation du temps d'arrivée au dépôt), ainsi que les détails que nous avons jugés nécessaires pour la compréhension de notre approche. Le chapitre suivant présentera des résultats expérimentaux de notre méthode.

Chapitre 5 : Résultats numériques

Nous allons présenter dans ce chapitre les résultats concernant des instances connues dans la littérature du TSPTW et d'autres instances issues d'un cas pratique que nous avons traité. Commençons par les résultats concernant les instances issues de la littérature du TSPTW.

5.1 Résultats pour des instances de la littérature du TSPTW

5.1.1 Description des instances

Plusieurs instances sont connues dans la littérature du TSPTW. Nous pouvons trouver ces instances, ainsi que leurs descriptions à la page web http://lopez-ibanez.eu/tsptw-instances. Nous considérons les groupes d'instances suivants :

1 Le groupe d'instances AFG contient 50 instances asymétriques. Ces instances sont issues d'un cas réel et ont été présentées par Ascheuer (1995). Le nombre de clients de ces instances varie de 10 à 233.

2 Le groupe d'instances DUMAS contient 135 instances qui sont regroupées en 27 groupes (contenant cinq instances chacun). Un groupe d'instances n20w40, par exemple, contient cinq instances (nommées n20w40.001, n20w40.002, n20w40.003, n20w40.004 et n20w40.005) qui contiennent chacune 20 clients, et pour chaque client, la largeur maximale de la fenêtre de temps est 40 unités de temps. Ces instances ont été proposées par Dumas et al. (1995). Le nombre de clients de ces instances varie de 20 à 200.

3 Le groupe d'instances GENDREAU contient 130 instances qui sont regroupées en 26 groupes. Ces instances ont été présentées par Gendreau et al. (1998) et elles étaient déduites en augmentant les largeurs des fenêtres de temps de quelques instances DUMAS. Le nombre de clients varie de 20 à 100.

4 Le groupe d'instances OHLMANN contient 25 instances. Ces instances ont été présentées par Ohlmann et Thomas (2007) et elles étaient déduites en augmentant

les largeurs des fenêtres de temps de quelques instances DUMAS. Le nombre de clients de ces instances varie de 150 à 200.

5.1.2 Résultats

Pour tous les résultats présentés, le paramètre k_{max} (voir Figure 7) a été fixé à 60. En effet, nous avons testé les valeurs 30, 45, 60 et 75 pour le paramètre k_{max}. Après quelques tests expérimentaux, nous avons remarqué que les valeurs 60 et 75 donnent des résultats meilleurs, vis-à-vis de la meilleure solution trouvée, que les valeurs 30 et 45. Cependant, la valeur 75 n'a pas amélioré les résultats trouvés avec la valeur 60. De plus, les temps d'exécutions correspondant à la valeur 60 (k_{max}=60) ont été légèrement inférieurs à ceux correspondant à la valeur 75.

Notre programme est séquentiel et a été exécuté sur un processeur Intel(R) Core(TM) i7-3770 dont la fréquence de l'horloge est de 3,40 GHz et la mémoire cache est de 8 Mo. Le langage de programmation est le C++. Dans cette partie, les résultats concernant les temps d'exécution sont tous multipliés par un facteur égal à 2,5. Nous expliquerons par la suite la raison de cette multiplication par 2,5.

Nous allons comparer les résultats obtenus par notre méthode à ceux obtenus par la méthode Beam-ACO (López-Ibáñez et al., 2013) qui est considérée comme l'état de l'art actuellement pour le TSPTW-C (minimisation du temps d'arrivée au dépôt). Cependant, les résultats présentés dans López-Ibáñez et al. (2013) correspondent à l'exécution sur un processeur Intel Xeon E5410 dont la fréquence de l'horloge est de 2,33 GHz et la mémoire cache est de 6 Mo. Le langage de programmation utilisé pour la méthode Beam-ACO est également le C++. Pour pouvoir comparer les temps d'exécution des deux méthodes, nous nous sommes basés sur des mesures de performances présentées par SPEC (pour *Standard Performance Evaluation Corporation*, qui est une organisation qui vise à maintenir à jour des mesures de performances standardisées pour différents processeurs, voir http://spec.org/cpu2006/results/cint2006.html). Ces mesures indiquent que le processeur que nous avons utilisé est plus performant (avec un facteur variant de 2,33 à 2,71), vis-à-vis du temps d'exécution, que celui utilisé par López-Ibáñez et al. (2013). Ainsi nous avons décidé de multiplier les temps d'exécution de notre méthode par

un facteur égal à 2,5. Les limites de temps t_{max} (voir Figure 7) ont été divisés par le même coefficient 2,5. Le paramètre t_{max} est donc choisi égal à 24 s lors de chaque exécution, car la limite de temps imposée dans la méthode Beam-ACO est de 60 s. Notons que pour les comparaisons des temps rapportés dans les prochains tableaux, nous considérons que les temps sont égaux pour les deux méthodes si la différence ne dépasse pas une seconde.

Pour la méthode Beam-ACO, des tests rigoureux de préréglage ont été effectués, pour 20% des instances de chaque groupe d'instances (AFG, DUMAS, GENDREAU et OHLMANN), pour déterminer des configurations (i.e., des choix des paramètres) qui donnent de bons résultats pour le groupe d'instances en question. De plus, après les tests de préréglage, les différentes configurations identifiées dans le préréglage sont testées sur les 80% des instances non utilisées dans les tests de préréglage, puis les auteurs rapportent les résultats de la meilleure configuration pour chaque groupe d'instances. Quant à notre méthode, elle est robuste et le paramètre k_{max} a été choisi égal à 60 pour tous les groupes d'instances.

Les résultats présentés suivent la même forme que ceux présentés dans López-Ibáñez et al. (2013). Pour chaque instance : le champ *Best* indique la meilleure valeur connue pour cette instance lorsque l'objectif est de minimiser le temps d'arrivée au dépôt destination ; le champ *%inf* représente le pourcentage de fois où la méthode Beam-ACO n'a pas pu identifier une solution réalisable pour l'instance. Ce champ n'est pas indiqué pour notre méthode GVNS, car elle arrive à trouver une solution réalisable pour toutes les exécutions de toutes les instances testées ; les champs *ERm* et *ERet* représentent, respectivement, la moyenne et l'écart-type de l'erreur relative *ER* ($ER = \frac{valeur-best}{best}$ où *valeur* représente la valeur retournée par une exécution donnée) ; les champs *Tm* et *Tet* sont en secondes et représentent, respectivement, la moyenne et l'écart-type du temps d'exécution qui a été nécessaire pour trouver la meilleur valeur retournée par une exécution donnée.

Pour chaque instance, nous exécutons notre méthode 15 fois, comme dans López-Ibáñez et al. (2013), et nous rapportons les résultats trouvés.

La notation NR signifie que les résultats n'étaient pas rapportés par López-Ibáñez et al. (2013). Les valeurs en gras indiquent que, au mieux de nos connaissances, les résultats concernant l'instance en question, pour le TSPTW-C, ne sont pas rapportés dans la littérature,

et qu'ils correspondent donc aux résultats de notre méthode GVNS (comme indiqué dans López-Ibáñez et al. (2013), ceci en soi est une contribution à la littérature du TSPTW-C). Les valeurs en gras et soulignées indiquent une amélioration de la meilleure valeur connue pour une instance.

Le Tableau II présente les résultats des instances AFG pour les méthodes Beam-ACO et GVNS. Nous rappelons que les résultats rapportés pour Beam-ACO correspondent à la configuration qui a donné les meilleurs résultats pour ce groupe d'instances. Beam-ACO trouve la meilleure solution connue pour toutes les instances (toutes les instances rapportées par López-Ibáñez et al. (2013)) à part pour les instances "rbg193.tw" et "rbg233.tw" où elle ne trouve pas une solution réalisable dans, respectivement, 6.67% et 20% des cas. Dans les autres cas la méthode trouve la meilleure solution connue. Les résultats de dix instances (20% des instances AFG) n'ont pas été rapportés car ces instances étaient utilisées dans l'étape de préréglage des paramètres (les résultats ne sont pas affichés également sur la page web http://lopez-ibanez.eu/tsptw-instances qui contient les instances et qui est supposée être tenue à jour, par deux des auteurs de López-Ibáñez et al. (2013). Pour indiquer les meilleures solutions connues actuellement pour les variantes TSPTW-T et TSPTW-C), nous fournissons les résultats trouvés par notre méthode pour ces instances. La méthode GVNS arrive à trouver la meilleure solution connue pour chaque instance dans chacune des 15 exécutions, à part pour l'instance "rbg172a.tw", où elle trouve la meilleure solution connue dans 13 parmi les 15 exécutions. GVNS est également plus rapide que Beam-ACO pour les instances AFG (voir les résultats des instances rbg021.6.tw, rbg172a.tw, rbg193.tw, rbg201a.tw et rbg233.tw).

			Beam-ACO					GVNS			
Instance	**Best**	**%inf**	**ERm**	**ERet**	**Tm**	**Tet**		**ERm**	**ERet**	**Tm**	**Tet**
rbg010a.tw	**3840**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg016a.tw	**2596**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg016b.tw	2094	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg017.2.tw	2351	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg017a.tw	4296	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg017.tw	**2351**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg019a.tw	2694	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg019b.tw	**3840**	NR	NR	NR	NR	NR		0,00	0,00	0	0

rbg019c.tw	**4536**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg019d.tw	3479	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg020a.tw	4689	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg021.2.tw	4528	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg021.3.tw	4528	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg021.4.tw	**4525**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg021.5.tw	4516	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg021.6.tw	4492	0.00	0.00	0.00	3	3		0,00	0,00	0	0
rbg021.7.tw	4481	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg021.8.tw	4481	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg021.9.tw	4481	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg021.tw	**4536**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg027a.tw	5093	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg031a.tw	3498	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg033a.tw	3757	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg034a.tw	3314	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg035a.2.tw	3325	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg035a.tw	3388	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg038a.tw	5699	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg040a.tw	5679	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg041a.tw	3793	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg042a.tw	3260	0.00	0.00	0.00	1	1		0,00	0,00	1	1
rbg048a.tw	9799	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg049a.tw	13257	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg050a.tw	**12050**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg050b.tw	11957	0.00	0.00	0.00	1	1		0,00	0,00	0	0
rbg050c.tw	10985	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg055a.tw	6929	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg067a.tw	10331	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg086a.tw	16899	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg092a.tw	**12501**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg125a.tw	14214	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg132.2.tw	18524	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg132.tw	18524	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg152.3.tw	**17455**	NR	NR	NR	NR	NR		0,00	0,00	0	0
rbg152.tw	17455	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg172a.tw	17783	0.00	0.00	0.00	18	19		0,00	0,00	11	13
rbg193.2.tw	21401	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg193.tw	21401	6.67	0.00	0.00	16	14		0,00	0,00	0	0

rbg201a.tw	21380	0.00	0.00	0.00	8	7		0,00	0,00	0	0
rbg233.2.tw	26143	0.00	0.00	0.00	0	0		0,00	0,00	0	0
rbg233.tw	26143	20.00	0.00	0.00	14	9		0,00	0,00	0	0

Tableau II. Les résultats des instances AFG

Le Tableau III présente les résultats des instances DUMAS pour Beam-ACO et GVNS. Les résultats sont regroupés pour chaque classe d'instance (chaque classe d'instance contient cinq instances). Beam-ACO trouve la meilleure solution connue pour toutes les classes d'instances (toutes les classes d'instances reportées par López-Ibáñez et al. (2013)) à chaque exécution, à part pour les classes d'instance n150w60[1] et n200w40. Pour la classe d'instances n200w40 Beam-ACO ne trouve pas une solution réalisable dans 8% des cas (ce qui correspond à 6 parmi 75 exécutions, car chacune des cinq instances est exécutée 15 fois; les auteurs ne fournissent pas d'informations sur la ou les instances que Beam-ACO n'a pas pu résoudre). Notons que, pour Beam-ACO, les résultats rapportés incluent les résultats des instances utilisées pour le préréglage (les auteurs ont utilisé dans le préréglage une instance de chaque classe d'instance testée), et les auteurs n'ont donné aucune raison pour laquelle ils n'ont pas rapporté les résultats de quelques classes d'instances. En ce qui concerne notre méthode GVNS, elle arrive à trouver la meilleure solution connue pour toutes les instances à chaque exécution à part pour l'instance "n60w100.004", où elle n'arrive jamais à trouver la meilleure solution connue pour cette instance. De plus, GVNS arrive à améliorer la meilleure valeur connue pour les deux classes d'instances n80w40 et n80w80. Les anciennes meilleures valeurs (rapportées par Carlton et Barnes (1996)) étaient, respectivement, 726 et 715,2. Les nouvelles meilleures valeurs identifiées par notre méthode sont 725,6 et 714,6, ce qui indique que le nombre des instances pour lesquelles nous avons identifié une nouvelle meilleure valeur est au moins égal à deux et au plus égal à cinq (car chaque classe contient cinq instances et la valeur de la solution de chaque

[1] Nous pensons qu'il y a une erreur typographique dans López-Ibáñez et al. (2013) pour le résultat de la classe d'instances n150w60 qui est reporté égal à 981,4 alors que dans la page web de deux des auteurs du dit article les résultats correspondent à 988,6. Notons également que la meilleure solution trouvée par notre méthode et par Carlton et Barnes (1996) est 988,6. Ainsi, il est possible que Beam-ACO a trouvé la meilleure solution connue 988,6 mais que les auteurs ont indiqué que Beam-ACO n'a pas pu trouver la meilleure solution connue.

instance est entière). GVNS est plus rapide que Beam-ACO pour quelques classes d'instances (n150w40, n150w60, n200w20 et n200w40).

		Beam-ACO						GVNS		
Instance	**Best**	**%inf**	**ERm**	**ERet**	**Tm**	**Tet**	**ERm**	**ERet**	**Tm**	**Tet**
n20w20	370,4	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n20w40	342,8	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n20w60	362	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n20w80	363,4	NR	NR	NR	NR	NR	0,00	0,00	0	0
n20w100	331,6	NR	NR	NR	NR	NR	0,00	0,00	0	0
n40w20	521,2	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w40	512,2	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w60	481,4	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w80	486,6	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w100	463	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n60w20	626,8	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n60w40	654,4	NR	NR	NR	NR	NR	0,00	0,00	0	0
n60w60	672,8	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n60w80	628,2	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n60w100	620,2	0,00	0,00	0,00	0	0	0,06	0,00	0	0
n80w20	748,2	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n80w40	**725,6**	NR	NR	NR	NR	NR	0,00	0,00	0	0
n80w60	712,6	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n80w80	**714,6**	NR	NR	NR	NR	NR	0,00	0,00	0	0
n100w20	823	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n100w40	821	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n100w60	817,2	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n150w20	978,4	0,00	0,00	0,00	1	1	0,00	0,00	0	0
n150w40	990,4	0,00	0,00	0,00	4	4	0,00	0,00	0	0
n150w60	988,6	0,00	0,73	0,00	4	4	0,00	0,00	1	0
n200w20	1137,8	0,00	0,00	0,00	7	8	0,00	0,00	1	1
n200w40	1156	8,00	0,00	0,00	15	13	0,00	0,00	1	1

Tableau III. Les résultats des instances DUMAS

Le Tableau IV présente les résultats des méthodes Beam-ACO et GVNS pour les instances GENDREAU. La meilleure configuration de Beam-ACO pour ce groupe d'instances arrive toujours (pour les classes d'instances rapportées par López-Ibáñez et al. (2013)) à trouver la meilleure valeur connue. GVNS arrive à trouver la meilleure solution connue pour chaque

classe d'instances. Cependant, pour cinq groupes d'instances (précisément, pour six instances) elle arrive à trouver la meilleure valeur connue, mais pas à chaque exécution. Nous fournissons également les résultats de cinq groupes d'instances pour la première fois, au mieux de nos connaissances, dans la littérature du TSPTW-C, les auteurs de López-Ibáñez et al. (2013) n'expliquent pas pourquoi les résultats de ces instances n'ont pas été rapportés. Notons également que les résultats rapportés pour Beam-ACO contiennent les résultats des instances utilisées dans l'étape de préréglage. Les temps nécessaires pour identifier les meilleures solutions pour chaque méthode sont quasi identiques (nous rappelons que nous ne considérons pas les différences de temps inférieures ou égales à une seconde).

		Beam-ACO						GVNS		
Instance	**Best**	**%inf**	**ERm**	**ERet**	**Tm**	**Tet**	**ERm**	**ERet**	**Tm**	**Tet**
n20w120	319,6	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n20w140	286,2	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n20w160	311,4	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n20w180	**311,2**	NR	NR	NR	NR	NR	0,00	0,00	0	0
n20w200	281,8	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w120	470,6	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w140	458,2	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w160	**426,8**	NR	NR	NR	NR	NR	0,00	0,00	0	0
n40w180	427,4	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n40w200	**412**	NR	NR	NR	NR	NR	0,00	0,00	0	0
n60w120	573,8	0,00	0,00	0,00	0	0	0,05	0,08	1	1
n60w140	600	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n60w160	619,6	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n60w180	576	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n60w200	570,2	0,00	0,00	0,00	0	0	0,02	0,02	0	1
n80w100	**711,2**	NR	NR	NR	NR	NR	0,00	0,00	0	0
n80w120	**697,4**	NR	NR	NR	NR	NR	0,01	0,01	1	1
n80w140	672,8	0,00	0,00	0,00	0	0	0,00	0,00	1	1
n80w160	653,6	0,00	0,00	0,00	1	1	0,23	0,24	2	2
n80w180	656,4	0,00	0,00	0,00	1	1	0,05	0,09	1	1
n80w200	646,2	0,00	0,00	0,00	1	0	0,00	0,00	2	3
n100w80	805,8	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n100w100	795,8	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n100w120	895,4	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n100w140	906,4	0,00	0,00	0,00	0	0	0,00	0,00	0	0
n100w160	865	0,00	0,00	0,00	0	0	0,00	0,00	0	0

Tableau IV. Les résultats des instances GENDREAU

Le Tableau V présente les résultats des méthodes Beam-ACO et GVNS pour les instances OHLMANN. Beam-ACO trouve la meilleure valeur connue pour toutes les instances rapportées dans López-Ibáñez et al. (2013) (les résultats de 20% des instances n'étaient pas rapportés, car ces instances ont été utilisées dans l'étape de préréglage). Les auteurs n'indiquent pas si la meilleure solution connue est trouvée à chaque exécution (la valeur moyenne de l'erreur relative est rapportée avec une précision de 0.01, ce qui ne nous permet pas d'assurer si la meilleure valeur connue a été trouvée pour chaque exécution). Toutefois, nous remarquons que les résultats de Beam-ACO sont bons pour les instances OHLMANN. En considérant les instances rapportées par López-Ibáñez et al. (2013) -pour pouvoir comparer les deux méthodes, GVNS arrive à trouver la meilleure valeur connue pour toutes les instances à chaque exécution, à part pour l'instance "n200w120.1" où GVNS arrive à trouver la meilleure valeur connue dans 13 parmi 15 exécutions. GVNS est plus rapide que Beam-ACO pour les instances "n200w120.5", "n200w140.3" et "n200w140.4", tandis que Beam-ACO est plus rapide que GVNS pour les instances "n150w120.1", "n200w120.1", "n200w120.3" et "n200w140.1". Pour les cinq instances non reportées par López-Ibáñez et al. (2013), nous fournissons les résultats des meilleures valeurs connues pour la première fois de la littérature. Nous notons que GVNS trouve la meilleure valeur connue pour l'instance "n150w160.2" dans 11 parmi 15 exécutions et quelle n'arrive pas à trouver la meilleure solution connue pour l'instance "n150w120.3" (notons que cette meilleure valeur a été identifiée par notre méthode, mais pour un autre ordre des voisinages dans la procédures VND).

			Beam-ACO					**GVNS**			
Instance	**Best**	**%inf**	**ERm**	**ERet**	**Tm**	**Tet**		**ERm**	**ERet**	**Tm**	**Tet**
n150w120.1	972	0,00	0,00	0,00	1	1		0,00	0,00	3	3
n150w120.2	917	0,00	0,00	0,00	1	2		0,00	0,00	1	0
n150w120.3	**909**[2]	NR	NR	NR	NR	NR		0,40	0,28	11	11
n150w120.4	925	0,00	0,00	0,00	1	1		0,00	0,00	2	2
n150w120.5	907	0,00	0,00	0,00	1	1		0,00	0,00	2	1

[2] Cette meilleure valeur connue a été identifiée par notre méthode GVNS mais avec un autre ordre de la procédure VND, voir la section 4.2.2.2 notamment le Tableau I.

n150w140.1	**1008**	NR	NR	NR	NR	NR	0,00	0,00	1	0
n150w140.2	**1020**	NR	NR	NR	NR	NR	0,00	0,00	3	3
n150w140.3	844	0,00	0,00	0,00	0	0	0,00	0,00	1	0
n150w140.4	898	0,00	0,00	0,00	1	1	0,00	0,00	1	1
n150w140.5	926	0,00	0,00	0,00	0	0	0,00	0,00	1	0
n150w160.1	959	0,00	0,00	0,00	1	1	0,00	0,00	1	0
n150w160.2	**890**	NR	NR	NR	NR	NR	0,03	0,05	18	15
n150w160.3	934	0,00	0,00	0,00	1	1	0,00	0,00	1	0
n150w160.4	**912**	NR	NR	NR	NR	NR	0,00	0,00	1	0
n150w160.5	920	0,00	0,00	0,00	0	0	0,00	0,00	1	0
n200w120.1	1089	0,00	0,00	0,00	9	8	0,28	1,04	13	11
n200w120.2	1072	0,00	0,00	0,00	1	2	0,00	0,00	2	1
n200w120.3	1128	0,00	0,00	0,00	4	5	0,00	0,00	7	6
n200w120.4	1072	0,00	0,00	0,00	4	5	0,00	0,00	3	1
n200w120.5	1073	0,00	0,00	0,00	5	5	0,00	0,00	2	0
n200w140.1	1138	0,00	0,00	0,00	12	12	0,00	0,00	19	17
n200w140.2	1087	0,00	0,00	0,00	4	5	0,00	0,00	3	0
n200w140.3	1083	0,00	0,00	0,00	12	11	0,00	0,00	5	3
n200w140.4	1100	0,00	0,00	0,00	12	8	0,00	0,00	10	8
n200w140.5	1121	0,00	0,00	0,00	5	9	0,00	0,00	5	2

Tableau V. Les résultats des instances OHLMANN

Nous avons présenté les résultats numériques pour quelques groupes d'instances connues pour le TSPTW en les comparant aux résultats obtenus par la méthode Beam-ACO considérée comme l'état de l'art actuellement pour le TSPTW-C. Récapitulons les résultats des comparaisons en tenant en compte la qualité des solutions obtenues, le pourcentage des solutions réalisables trouvées et le temps nécessaire pour trouver les meilleures solutions connues :

- GVNS est plus performante que Beam-ACO pour le groupe d'instances AFG ;
- GVNS et Beam-ACO sont compétitives pour le groupe d'instances DUMAS;
- GVNS est moins performante que Beam-ACO pour le groupe d'instances GENDREAU (vis-à-vis du pourcentage des exécutions où on trouve la meilleure solution connue) ;
- GVNS et Beam-ACO sont compétitives pour le groupes d'instances OHLMANN (avec un léger avantage de Beam-ACO pour l'instance n200w120.1) ;

- GVNS est robuste, les mêmes paramètres ont été utilisés pour tous les groupes d'instances ;
- Les résultats rapportés pour Beam-ACO correspondent à des configurations différentes dépendamment du groupe d'instances traité ;
- Beam-ACO semble être mieux adaptée aux instances avec de larges fenêtres de temps; cependant, elle n'arrive pas à trouver des solutions réalisables pour quelques instances avec des fenêtres de temps relativement petites (en considérant la meilleure configuration pour ces groupes d'instances).
- Pour les groupes d'instances DUMAS et GENDREAU, les résultats de quelques classes d'instances ne sont pas rapportés par (López-Ibáñez et al., 2013) sans aucune explication de la part des auteurs.

Nous avons présenté les principaux points de différence entre la méthode Beam-ACO et notre méthode GVNS, mais nous ne pouvons pas trancher laquelle des deux est la plus performante. Nous pensons que cela dépend des champs d'application et de la structure des instances. Cependant, nous pouvons assurer que notre méthode est compétitive avec l'état de l'art.

5.2 Résultats pour le cas pratique traité

Pour le contexte du cas pratique, nous référons le lecteur à la section 2.1.1.

L'objectif de l'entreprise, qui est spécialisée dans le développement de solutions logicielles intégrées pour la gestion du transport et de la livraison du courrier et des colis, est de minimiser la durée totale d'une tournée (TSPTW-D). Grâce à l'outil performant de l'entreprise qui fournit le temps de départ du dépôt origine, nous avons pu ramener ce problème à la résolution du TSPTW-C qui est moins compliqué que le TSPTW-D. L'algorithme actuel de l'entreprise est déterministe (il trouve la même solution pour une instance donnée). Nous allons comparer les résultats (au niveau des durées des tournées) entre notre algorithme et l'algorithme

actuel de l'entreprise. Le paramètre k_{max} (voir la Figure 7) est choisi égal à 60, et la limite du temps t_{max} est choisie pour chaque instance comme étant le temps d'exécution de l'algorithme actuel de l'entreprise. Les temps présentés dans cette partie sont les vrais temps d'exécution (sans les multiplier par aucun facteur), et l'algorithme actuel de l'entreprise est exécuté sur un processeur Intel(R) Core(TM) i7-4790 dont la fréquence de l'horloge est 3,60 GHz et la mémoire cache est de 8 Mo. Nous considérons que ce processeur est comparable à celui que nous avons utilisé (Intel(R) Core(TM) i7-3770 dont la fréquence de l'horloge est de 3,40 GHz et la mémoire cache est de 8 Mo).

Nous disposons des données de six instances du TSPTW (en réalité ces six instances correspondent à une seule instance du VRPTW). Nous exécutons chaque instance dix fois. Le Tableau VI présente les résultats comparatifs entre l'algorithme actuel de l'entreprise (nommé *Algorithme1*) et notre algorithme GVNS. Le champ *n* indique le nombre de clients de chaque instance; le champ *Durée* indique la valeur de la solution obtenue par l'algorithme actuel de l'entreprise (une durée sous la forme 07:15:40, par exemple, représente une durée de 7 heures 15 minutes et 40 secondes); le champ *Te* représente le temps d'exécution, en secondes, de l'algorithme de l'entreprise (ce temps est celui que nous considérons comme limite de temps pour notre algorithme GVNS; notons que ce temps peut être différent du temps nécessaire pour identifier la meilleure solution); le champ *DuréeMin* indique la meilleure durée trouvée par notre algorithme pour dix exécutions; le champ *DuréeMax* indique la pire durée trouvée par notre algorithme pour dix exécutions; le champ *DuréeMoy* indique la moyenne des durées trouvées par notre algorithme pour dix exécutions et le champ *Tm* représente le temps moyen en secondes, pour dix exécutions, nécessaire pour trouver la meilleure solution à chaque exécution. La dernière ligne du tableau calcule l'amélioration totale (nous rappelons qu'en réalité, ces 6 instances correspondent à une seule instance du VRPTW) des durées trouvées par notre algorithme par rapport à celles identifiées par l'algorithme actuel de l'entreprise. Nous remarquons que pour chaque instance, notre algorithme améliore (strictement) les résultats de l'entreprise pour chacune des dix exécutions. Nous indiquons également que, pour l'instance numéro 4, la solution trouvée par l'algorithme actuel de l'entreprise n'est pas réalisable; en fait, le temps d'arrivée à un seul client est 17 secondes plus tard que la fin de la fenêtre horaire du dit client. En résumé, dans le scénario le plus optimiste (pour les résultats rapportés), l'amélioration

totale des durées des six tournées est de 32 minutes et 19 secondes, l'amélioration totale dans le scénario le plus pessimiste est de 25 minutes et 28 secondes et l'amélioration totale, en moyenne, est de 29 minutes et 38 secondes. Les solutions de toutes les exécutions de notre algorithme sont réalisables, et les temps nécessaires pour identifier la meilleure solution à chaque exécution sont jugés satisfaisants.

	Algorithme1				**GVNS**			
Instance	n	Durée	Te		DuréeMin	DuréeMax	DuréeMoy	Tm
Instance1	100	07:15:40	44		07:02:01	07:04:27	07:03:13	10
Instance2	93	05:47:22	15		05:45:03	05:47:01	05:45:50	8
Instance3	105	06:16:16	10		06:09:07	06:11:34	06:09:50	3
Instance4	75	06:13:26	11		06:04:44	06:04:44	06:04:44	2
Instance5	22	01:46:05	< 1		01:46:02	01:46:02	01:46:02	0
Instance6	35	02:36:43	< 1		02:36:16	02:36:16	02:36:16	0
Amélioration totale					00:32:19	00:25:28	00:29:38	

Tableau VI. Les résultats des instances du cas pratique

Conclusion

Le point de départ de cette étude est un cas pratique de minimisation de la durée totale d'un problème de voyageur de commerce avec fenêtres de temps. Nous avons ramené ce problème à un problème moins compliqué qui vise la minimisation du temps d'arrivée au dépôt destination pour le problème de voyageur de commerce avec fenêtres de temps.

Nous avons adapté un algorithme qui a été présenté pour la minimisation de la distance d'une tournée pour le TSPTW. Nous avons introduit (pour l'algorithme à adapter) des tests d'élimination des arcs incompatibles et nous avons présenté comment profiter de ces tests pour réduire l'espace de recherche. Nous avons présenté notamment un processus efficient pour la vérification de la rentabilité d'un mouvement en le combinant à un processus existant de vérification de la réalisabilité et un bon ordre d'exploration d'un voisinage. La méthode présentée est une méthode à deux phases. La première phase trouve une solution réalisable à l'aide d'un algorithme de type VNS, et la deuxième phase utilise une heuristique GVNS pour améliorer la solution identifiée à la première phase.

Les résultats expérimentaux indiquent que notre méthode est robuste, ce qui représente un grand avantage pour les applications considérant des instances diversifiées, et est compétitive avec l'état de l'art. Nous avons amélioré les meilleures solutions connues pour deux classes d'instances et nous avons fourni pour la première fois, au mieux de nos connaissances, les résultats de plusieurs instances, connues dans la littérature du TSPTW, lorsque la fonction objectif est le temps d'arrivée au dépôt destination (la variante appelée dans ce document TSPTW-C). Les résultats obtenus pour le cas pratique considéré sont jugés satisfaisants.

Bibliographie

Ascheuer, N. (1995). Hamiltonian path problems in the on-line optimization of flexible manufacturing systems. *Ph.D.Thesis, Technische Universität Berlin, Germany*.

Baker, E. K. (1983). Technical note—an exact algorithm for the time-constrained traveling salesman problem. *Operations Research, 31*(5), 938-945.

Baldacci, R., Mingozzi, A. et Roberti, R. (2012). New state-space relaxations for solving the traveling salesman problem with time windows. *INFORMS Journal on Computing, 24*(3), 356-371.

Blum, C. (2005). Beam-ACO—Hybridizing ant colony optimization with beam search: An application to open shop scheduling. *Computers & Operations Research, 32*(6), 1565-1591.

Calvo, R. W. (2000). A new heuristic for the traveling salesman problem with time windows. *Transportation Science, 34*(1), 113-124.

Carlton, W. B. et Barnes, J. W. (1996). Solving the traveling-salesman problem with time windows using tabu search. *IIE transactions, 28*(8), 617-630.

Christofides, N., Mingozzi, A. et Toth, P. (1981). State-space relaxation procedures for the computation of bounds to routing problems. *Networks, 11*(2), 145-164.

Da Silva, R. F. et Urrutia, S. (2010). A General VNS heuristic for the traveling salesman problem with time windows. *Discrete Optimization, 7*(4), 203-211.

Desrosiers, J., Dumas, Y., Solomon, M. M. et Soumis, F. (1995). Time constrained routing and scheduling. *Handbooks in operations research and management science, 8*, 35-139.

Dumas, Y., Desrosiers, J., Gelinas, E. et Solomon, M. M. (1995). An optimal algorithm for the traveling salesman problem with time windows. *Operations research, 43*(2), 367-371.

Favaretto, D., Moretti, E. et Pellegrini, P. (2006). An ant colony system approach for variants of the traveling salesman problem with time windows. *Journal of information and optimization sciences, 27*(1), 35-54.

Focacci, F., Lodi, A. et Milano, M. (2002). A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing, 14*(4), 403-417.

Funke, B., Grünert, T. et Irnich, S. (2005). Local search for vehicle routing and scheduling problems: Review and conceptual integration. *Journal of heuristics, 11*(4), 267-306.

Gendreau, M., Hertz, A., Laporte, G. et Stan, M. (1998). A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research, 46*(3), 330-335.

Hansen, P., Mladenović, N. et Pérez, J. A. M. (2008). Variable neighbourhood search: methods and applications. *4OR, 6*(4), 319-360.

Ilavarasi, K. et Joseph, K. S. (2014). *Variants of travelling salesman problem: A survey. 2014 International Conference on Information Communication and Embedded Systems (ICICES),* 1-7.

Kara, I., Koc, O. N., Altıparmak, F. et Dengiz, B. (2013). New integer linear programming formulation for the traveling salesman problem with time windows: minimizing tour duration with waiting times. *Optimization, 62*(10), 1309-1319.

Kona, H., Burde, A. et Zanwar, D. (2015). A Review of Traveling Salesman Problem with Time Window Constraint. *International Journal for Innovative Research in Science and Technology, 2*(1), 253-256.

Langevin, A., Desrochers, M., Desrosiers, J., Gélinas, S. et Soumis, F. (1993). A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows. *Networks, 23*(7), 631-640.

Li, J.-Q. (2009). A Bi-directional Resource-bounded Dynamic Programming Approach for the Traveling Salesman Problem with Time Windows. *Submitted manuscript*.

Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal, The, 44*(10), 2245-2269.

Lin, S. et Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations research, 21*(2), 498-516.

López-Ibáñez, M. et Blum, C. (2010). Beam-ACO for the travelling salesman problem with time windows. *Computers & operations research, 37*(9), 1570-1583.

López-Ibáñez, M., Blum, C., Ohlmann, J. W. et Thomas, B. W. (2013). The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization. *Applied Soft Computing, 13*(9), 3806-3815.

Mladenović, N. et Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research, 24*(11), 1097-1100.

Mladenovic, N., Todosijevic, R. et Urosevic, D. (2012). An efficient general variable neighborhood search for large travelling salesman problem with time windows. *Yugoslav Journal of Operations Research ISSN: 0354-0243 EISSN: 2334-6043, 23*(1), 19-30.

Ohlmann, J. W. et Thomas, B. W. (2007). A compressed-annealing heuristic for the traveling salesman problem with time windows. *INFORMS Journal on Computing, 19*(1), 80-90.

Pesant, G., Gendreau, M., Potvin, J.-Y. et Rousseau, J.-M. (1998). An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science, 32*(1), 12-29.

Savelsbergh, M. W. (1985). Local search in routing problems with time windows. *Annals of Operations research, 4*(1), 285-305.

Savelsbergh, M. W. (1992). The vehicle routing problem with time windows: Minimizing route duration. *ORSA journal on computing, 4*(2), 146-154.

Solomon, M. M. (1987). Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research, 35*(2), 254-265.

Solomon, M. M. et Desrosiers, J. (1988). Survey paper-time window constrained routing and scheduling problems. *Transportation science, 22*(1), 1-13.

Yurtkuran, A. et Emel, E. (2014). Efficient Constraint Handling in Electromagnetism-Like Algorithm for Traveling Salesman Problem with Time Windows. *The Scientific World Journal, 2014:* ID 871242.

Les nouvelles solutions rapportées

Nous présentons dans cet annexe les nouvelles solutions qui sont rapportées dans ce document. Notons que, pour les classes d'instances n80w40 et n80w80, nous allons présenter les solutions des cinq instances de chacune des classes car nous ne savons pas quelle(s) instance(s) nous avons améliorée(s). Pour chaque solution présentée, le premier élément et le dernier élément correspondent au dépôt.

Instances AFG

rbg010a.tw : 0 2 4 3 1 7 6 5 8 9 10 11.

rbg016a.tw : 0 2 1 4 7 6 3 8 5 9 12 11 10 13 14 15 16 17.

rbg017.tw : 0 2 1 6 5 4 3 13 12 11 10 8 7 9 14 15 16.

rbg019b.tw : 0 8 1 7 2 6 5 4 3 14 13 9 12 11 10 15 18 16 17 19 20.

rbg019c.tw : 0 1 2 3 4 5 6 8 7 9 10 11 12 13 14 15 16 17 18 19 20.

rbg021.4.tw : 0 1 2 3 4 8 7 5 10 11 12 9 6 14 15 16 19 17 18 13 20.

rbg021.tw : 0 1 2 3 4 5 6 8 7 9 10 11 12 13 14 15 16 17 18 19 20.

rbg050a.tw : 0 2 1 3 45 46 42 8 7 20 19 6 18 17 5 4 36 15 35 14 13 12 11 34 32 9 10 33 30 29 28 27 26 24 23 22 21 37 16 25 38 31 41 48 39 50 44 40 47 43 49 51.

rbg092a.tw : 0 2 1 3 6 5 9 4 10 8 7 11 12 14 13 17 16 20 21 19 15 18 22 24 26 25 23 30 29 28 27 34 33 31 37 32 38 36 39 35 42 41 40 48 44 47 46 45 43 52 51 50 49 54 53 56 58 57 62 55 63 59 64 61

60 68 70 66 65 76 69 74 67 72 84 71 78 80 73 75 79 77 82 81 87 86 83 89 85 88 90 91 92 93.

rbg152.3.tw : 0 11 10 14 9 7 4 3 2 1 24 13 8 5 23 6 27 26 25 22 12 21 20 19 18 17 16 15 32 34 31 28 30 37 38 39 36 35 41 29 44 42 33 46 45 55 54 52 49 53 51 40 56 43 59 58 60 57 48 47 50 71 70 69 68 67 66 65 64 73 63 62 61 78 77 76 75 74 72 81 80 79 91 84 92 83 82 95 94 93 88 86 100 85 96 105 106 104 90 87 89 99 101 112 102 103 98 97 111 108 118 117 109 124 125 110 121 123 126 107 120 119 114 116 113 122 115 138 136 135 137 139 133 132 131 129 128 127 130 145 134 146 143 140 148 149 141 144 147 142 150 151.

Instances DUMAS

n80w40.001 : 0 5 56 51 38 17 60 50 27 61 54 53 19 4 6 34 24 33 9 30 14 69 3 13 72 70 45 71 20 66 36 49 64 57 68 78 25 48 8 29 12 58 55 41 42 79 15 35 21 52 32 1 59 44 7 73 47 62 26 75 10 28 74 16 46 76 31 43 39 63 37 18 65 2 40 23 67 77 11 80 22 81.

n80w40.002 : 0 41 49 78 74 19 43 66 52 26 64 65 3 36 20 12 71 63 17 9 25 61 27 32 72 23 24 44 79 10 62 35 53 14 47 45 31 75 50 11 5 48 30 51 4 8 18 55 22 7 59 60 42 69 56 21 29 6 33 58 73 34 70 54 76 38 37 2 28 16 68 40 15 77 57 39 46 1 13 80 67 81.

n80w40.003 : 0 52 19 42 54 57 45 35 10 71 56 53 79 67 28 69 40 64 27 4 12 11 24 58 32 44 18 61 39 73 49 17 38 29 51 72 34 8 7 5 33 14 31 21 76 46 20 62 3 55 6 65 74 59 16 22 23 78 15 68 41 47 43 13 75 48 30 25 63 2 9 36 1 66 50 70 77 26 80 60 37 81.

n80w40.004 : 0 8 5 78 20 24 61 40 54 48 72 29 21 64 41 76 74 77 6 26 59 22 51 25 60 32 71 45 10 44 43 33 16 9 62 36 56 34 18 39 79 46 2 31 50 63 58 70 37 55 30 15 47 28 14 13 67 1 42 52 69 27 49 65 3 35 73 23 19 17 75 66 12 4 11 38 7 53 57 80 68 81.

n80w40.005 : 0 11 20 24 21 76 18 74 42 49 22 4 33 62 67 55 69 17 15 41 34 61 7 66 60 39 29 71 32 35 2 63 45 8 13 57 46 31 26 50 77 14 10 64 73 56 36 5 38 51 54 19 12 16 65 1 68 70 75 52 3 44 58 59 40 78 25 72 53 6 9 23 47 28 43 79 27 30 80 37 48 81.

n80w80.001 : 0 18 8 21 66 52 77 72 11 32 31 6 42 13 27 58 43 4 3 60 36 55 9 28 12 41 76 46 38 75 65 19 40 17 34 26 57 44 48 23 7 22 20 79 59 37 71 78 24 54 70 14 67 63 25 16 30 69 50 1 5 74 47 61 33 49 2 64 45 62 35 29 10 68 51 39 73 15 80 53 56 81.

n80w80.002 : 0 25 57 36 62 41 43 11 30 32 19 1 76 45 75 59 8 49 34 56 29 21 16 72 58 26 28 42 46 50 3 44 15 7 13 38 12 55 17 35 52 78 20 48 9 60 53 67 40 6 22 79 73 24 65 37 33 31 4 77 71 68 70 66 64 18 63 39 10 14 47 54 2 51 61 5 27 69 74 80 23 81.

n80w80.003 : 0 77 25 48 58 57 19 76 18 17 52 70 3 13 51 1 60 71 40 11 37 8 73 42 10 15 47 4 34 61 45 62 65 39 41 55 29 54 74 75 36 12 32 53 46 22 23 2 33 16 5 35 30 63 24 69 66 7 72 14 31 56 67 9 43 28 21 68 49 50 20 64 79 6 38 27 78 59 44 80 26 81.

n80w80.004 : 0 16 41 58 19 73 30 37 72 70 71 45 60 57 10 79 74 53 61 12 77 3 39 24 1 66 8 44 28 21 2 22 33 36 7 76 32 20 56 40 23 31 65 52 49 15 51 18 64 63 48 59 35 69 62 14 27 38 34 17 42 25 78 6 11 43 26 67 55 9 50 47 46 4 68 54 75 13 5 80 29 81.

n80w80.005 : 0 16 73 33 5 72 45 79 4 77 50 44 71 9 31 48 55 63 61 47 24 57 21 26 62 53 52 65 68 32 54 60 13 3 76 10 27 46 56 74 40 43 28 8 58 69 7 14 64 15 49 78 34 22 37 75 30 67 29 18 1 35 19 66 20 6 39 2 11 70 12 25 41 36 38 17 59 23 80 42 51 81.

Instances GENDREAU

n20w180.001 : 0 19 13 3 12 15 1 17 5 10 11 14 7 16 18 6 4 8 20 2 9 21.

n20w180.002 : 0 16 1 12 13 9 6 17 5 3 10 14 15 18 11 7 4 19 8 20 2 21.

n20w180.003 : 0 15 3 9 16 19 13 14 10 1 6 11 4 2 18 7 5 12 20 17 8 21.

n20w180.004 : 0 18 10 3 17 8 19 2 7 11 4 1 14 9 6 5 13 20 15 16 12 21.

n20w180.005 : 0 3 9 19 1 8 16 10 6 5 14 7 4 12 18 13 11 15 2 17 20 21.

n40w160.001 : 0 15 10 33 6 36 8 16 17 4 7 19 32 25 31 1 27 9 22 18 26 21 12 14 13 23 3 5 35 30 34 28 38 29 2 39 20 40 11 24 37 41.

n40w160.002 : 0 38 28 8 14 6 30 20 3 18 34 26 24 15 27 1 10 35 13 5 33 9 37 2 17 23 29 25 12 21 19 16 11 4 31 40 39 36 32 7 22 41.

n40w160.003 : 0 6 18 34 36 27 8 35 10 12 16 19 28 26 1 39 33 22 25 23 4 17 31 32 9 37 30 14 38 24 20 2 15 7 29 5 3 13 21 11 40 41.

n40w160.004 : 0 13 26 19 33 3 7 37 18 30 9 28 15 4 14 29 31 24 36 38 32 6 39 11 5 8 23 10 27 2 20 35 16 25 34 21 1 22 40 12 17 41.

n40w160.005 : 0 4 20 2 37 24 15 28 36 1 17 18 25 11 23 27 3 14 29 9 5 6 7 39 30 8 22 21 33 31 13 35 26 16 38 19 10 32 12 40 34 41.

n40w200.001 : 0 27 26 1 16 5 34 20 31 19 8 14 2 13 33 32 18 21 3 36 24 37 9 22 29 4 12 30 15 25 35 10 39 11 40 7 17 38 6 28 23 41.

n40w200.002 : 0 4 36 8 26 22 24 11 15 10 39 25 37 9 34 12 33 1 30 32 23 28 21 29 2 3 7 19 16 14 6 18 38 5 17 27 35 31 13 40 20 41.

n40w200.003 : 0 22 25 2 23 34 17 26 16 13 30 12 14 10 18 8 15 3 39 21 35 31 7 4 29 1 24 28 38 27 20 9 6 33 11 37 5 36 40 32 19 41.

n40w200.004 : 0 18 24 39 26 19 6 10 17 29 28 16 37 22 2 3 11 15 1 20 27 36 14 32 7 21 5 8 40 33 23 38 34 13 30 9 12 25 35 31 4 41.

n40w200.005 : 0 1 25 37 24 4 12 29 20 6 14 28 34 26 39 7 16 13 33 15 21 27 2 22 31 19 36 10 30 23 32 3 18 5 35 17 8 11 9 40 38 41.

n80w100.001 : 0 61 67 7 22 10 15 31 2 27 58 5 77 54 37 48 30 4 39 60 74 73 46 23 29 32 24 76 6 47 78 63 21 18 62 12 51 75 11 3 72 34 14 50 68 42 36 1 55 16 25 71 45 26 65 9 43 49 70 28 59 53 35 44 33 20 57 41 64 8 38 40 56 13 52 79 69 66 17 80 19 81.

n80w100.002 : 0 47 7 69 39 35 32 46 44 56 40 50 48 60 25 58 38 76 51 70 67 13 34 3 42 41 37 77 66 31 18 11 2 59 53 65 30 4 8 78 14 52 26 79 49 45 74 57 6 10 28 71 62 9 22 15 54 43 72 75 5 27 24 12 33 1 19 63 20 16 21 61 64 73 29 68 55 36 80 23 17 81.

n80w100.003 : 0 28 64 47 51 59 54 5 18 1 37 9 38 32 69 42 52 78 67 72 40 74 7 76 35 50 39 44 62 15 17 41 56 79 27 21 10 14 33 65 60 25 31 2 71 66 24 3 36 23 75 22 30 61 45 13 73 43 48 70 49 46 57 26 29 34 63 53 11 77 12 6 20 58 16 55 80 4 68 8 19 81.

n80w100.004 : 0 60 62 34 22 8 10 29 54 2 48 13 35 51 44 66 45 77 31 20 12 18 73 56 33 16 41 75 61 15 11 76 53 24 28 6 3 1 67 79 4 40 68 27 49 65 52 63 50 55 21 17 14 39 59 74 64 25 70 9 38 42 36 30 37 7 69 32 19 72 5 58 78 47 26 43 57 46 71 80 23 81.

n80w100.005 : 0 57 45 50 43 36 1 15 25 19 33 49 30 39 38 72 11 69 64 16 77 23 20 14 66 76 18 59 46 60 79 74 5 8 52 47 73 44 42 61 27 21 17 31 48 54 12 6 7 62 9 24 53 32 34 22 78 4 75 67 13 51 3 63 2 58 40 28 71 26 68 10 37 70 29 55 65 35 80 41 56 81.

n80w120.001 : 0 17 58 29 77 79 53 64 39 37 31 34 19 50 38 9 2 33 6 1 48 8 3 54 74 71 30 76 73 66 7 49 14 15 5 55 56 44 22 24 35 25 32 18 65 52 20 60 62 47 21 13 61 4 46 41 16 28 67 68 57 59 40 23 43 72 78 36 45 12 51 70 27 26 11 42 69 10 75 80 63 81.

n80w120.002 : 0 73 5 30 64 36 38 61 22 11 31 18 21 16 3 13 58 48 2 53 19 9 6 32 57 1 8 52 44 74 56 23 51 29 24 66 20 41 43 35 27 37 7 40 49 55 26 42 4 76 50 15 34 71 63 17 79 72 47 25 12 78 45 33 46 70 69 75 59 39 60 67 28 14 65 54 68 10 62 80 77 81.

n80w120.003 : 0 41 45 52 46 11 50 8 66 34 44 48 10 6 3 69 13 24 61 2 17 20 75 9 30 36 25 73 56 51 72 49 39 59 77 42 58 43 55 64 78 4 53 76 74 57 28 31 14 35 12 40 70 16 37 15 60 71 62 26 1 67 18 27 22 65 19 79 33 21 23 7 47 38 54 32 29 63 5 80 68 81.

n80w120.004 : 0 47 24 70 20 16 65 14 12 74 63 9 39 62 10 8 53 5 72 22 25 75 11 29 23 67 31 42 34 64 51 56 73 45 40 43 15 21 61 36 44 3 30 48 17 79 38 18 19 37 1 60 66 6 57 35 41 27 68 49 77 2 50 69 76 7 28 58 59 52 54 46 32 71 55 4 26 78 80 13 33 81.

n80w120.005 : 0 58 66 39 64 71 9 63 22 4 16 12 70 47 42 78 50 54 69 5 61 27 11 20 6 31 48 55 59 46 53 18 37 75 74 2 62 49 68 34 7 15 28 73 25 65 32 10 67 19 41 38 35 76 51 44 52 45 17 1 26 43 36 40 30 29 3 79 77 57 8 23 21 13 60 14 72 56 24 80 33 81.

Instances OHLMANN

n150w120.3 : 0 64 82 131 40 28 148 78 7 140 141 137 97 129 93 27 29 32 68 11 36 34 94 38 30 53 98 19 24 89 54 84 5 51 26 149 44 126 31 65 83 81 85 9 70 123 33 103 76 107 142 69 104 3 52 17 86 79 116 59 46 37 50 22 63 100 39 102 42 18 99 73 4 120 57 14 128 115 91 61 55 2 10 113 62 96 75 108 106 43 146 139 90 143 41 136 58 118 87 15 110 8 114 117 20 23 25 147 74 1 109 144 135 112 67 60 92 45 21 127 111 56 130 125 48 145 88 6 133 13 72 71 119 80 66 105 95 12 122 132 101 47 138 16 124 77 134 121 150 49 35 151.

n150w140.1 : 0 149 102 142 40 30 109 80 72 112 12 19 147 124 117 8 138 105 35 20 15 41 133 103 61 86 46 116 118 48 55 113 28 120 23 36 63 1 2 13 71 96 108 64 106 37 104 140 32 9 93 38 27 21 50 82 57 77 25 54

145 92 73 29 87 49 137 111 33 144 66 5 134 16 22 91 146 60 69 10 123 34 45 131 119 6 68 44 81 101 67 97 88 65 121 83 99 130 17 58 26 79 141 39 136 100 139 74 62 115 24 56 47 84 76 95 43 52 42 148 85 31 14 70 128 125 114 110 127 132 90 7 135 59 143 98 18 126 51 78 129 75 11 94 89 3 107 4 150 122 53 151.

n150w140.2 : 0 148 120 22 32 70 146 107 9 39 64 126 125 59 94 105 143 57 90 117 140 17 23 40 8 97 48 58 37 41 5 27 71 38 67 3 135 101 108 114 19 110 102 85 75 36 46 60 31 52 43 127 88 6 53 98 106 73 134 30 91 74 109 103 139 116 130 11 123 49 54 69 144 10 122 20 61 79 13 33 82 12 145 81 62 35 87 149 78 147 92 86 121 80 34 2 7 72 138 55 76 18 1 132 104 83 96 26 65 118 63 100 66 137 133 119 93 112 21 77 111 128 84 68 51 28 56 141 95 25 24 15 131 44 47 142 89 42 113 14 129 115 124 50 45 136 150 16 99 29 4 151.

n150w160.2 : 0 148 92 89 51 35 44 63 73 48 133 41 86 99 19 4 97 142 61 12 139 42 109 38 81 21 11 56 88 114 136 140 121 22 71 95 102 116 85 24 3 77 138 16 145 70 26 122 147 96 104 66 50 34 69 67 115 27 93 59 87 2 107 30 31 39 112 105 132 37 68 144 118 25 80 100 74 17 62 119 18 131 29 98 10 72 58 52 128 120 60 149 8 1 82 78 6 113 84 103 53 106 83 134 76 57 32 54 127 75 137 7 94 33 143 5 101 129 9 124 111 49 110 126 90 13 125 135 141 43 14 36 108 46 65 47 55 40 45 91 15 146 64 20 79 28 23 123 117 130 150 151.

n150w120.3 : 0 144 131 129 44 39 137 113 4 92 3 26 77 114 98 11 46 43 136 41 140 87 147 42 59 62 81 149 38 76 75 84 7 135 12 35 8 66 32 83 74 148 49 45 145 37 122 103 55 48 21 107 112 70 125 72 18 36 118 123 117 34 16 73 64 105 79 100 67 23 86 143 139 124 95 94 54 142 119 138 127 47 15 28 146 25 128 89 101 102 133 56 6 65 17 9 63 68 60 93 78 69 141 108 71 96 14 121 97 50 13 27 99 58 110 85 29 2 40 132 30 53 1 24 57 126 109 134 120 52 33 130 19 91 61 5 115 106 90 88 51 10 104 20 116 22 80 82 111 31 150 151.