Master Parisien Recherches Opérationnelles

Rapport de projet ECMA

École nationale des Ponts et Chaussées
2024

Baptiste Vert
Élève ingénieur

1 Modélisation papier

Ici, le but du projet est d'étudier un problème de tournées de véhicules robuste. Le nombre de véhicules utilisé n'est ni contraint ni pénalisant.

Il y a n villes qui doivent être visitées une et une seule fois. On travaillera avec n camions (ce nombre pourra être amené à être réduit si la taille de l'instance à résoudre est trop importante), on cherche à déterminer un ensemble de tournées permettant de visiter l'ensemble des clients tout en minimisant la durée totale des trajets, qui correspond ici à la somme des temps de trajet entre chaque paire de ville présentes dans les tournées.

Pour la modélisation du problème, on souhaite travailler avec un modèle compact pour cela on va utiliser les inégalités dites MTZ, qui permettent d'éliminer la présence de sous-tours mais en ajoutant seulement un nombre polynomial de contraintes.

On notera pour $(i,j) \in A$, $k \in [n]$, x_{ijk} la variable binaire qui vaut 1 si le véhicule k dessert la ville j en partant de la ville i. On notera pour $i,k \in [n]$, u_{ik} la variable entière qui indique le nombre de villes visitées par le véhicule k en arrivant à la ville i et en ajoutant celle-ci.
Une modélisation statique du problème de tournée de véhicules est donc :

PLNE

Objectif :

$$\text{Min} \sum_{(i,j)\in A} \sum_{k=1}^{n} t_{ij} x_{ijk} \tag{1}$$

Sous contraintes :

$$\sum_{(i,j)\in A} \sum_{k=1}^{n} x_{ijk} = 1, \quad \forall j \in [\![2;n]\!]$$

$$\sum_{(i,j)\in A, j\neq 1} d_j x_{ijk} \leq C, \quad \forall k \in [\![1;n]\!]$$

$$\sum_{(1,j)\in A} x_{1jk} \geq \frac{1}{n^2} \sum_{(i,j)\in A, i\neq 1} x_{ijk}, \quad \forall k \in [\![1;n]\!]$$

$$\sum_{(i,1)\in A} x_{i1k} \geq \frac{1}{n} \sum_{(1,j)\in A, j\neq 1} x_{1jk}, \quad \forall k \in [\![1;n]\!]$$

$$x_{ijk} \leq \sum_{(l,i)\in A, l\neq i} x_{lik}, \quad \forall k,i,j \in [\![1;n]\!]$$

$$u_{jk} \geq u_{ik} + 1 - n(1 - x_{ijk}), \quad \forall k, j \in [\![1,n]\!], j \neq 1$$

$$x_{ijk} \in \{0,1\}, \quad \forall (i,j) \in A, \quad k \in [\![1,n]\!]$$

$$u_{ik} \in \mathbb{N}, \quad \forall i,k \in [\![1,n]\!]$$

Ici, chaque groupe de contraintes est associé à un véhicule sauf la première qui s'assure que chaque ville soit visitée une et une seule fois. La découpe du problème en n véhicules permet de ne travailler qu'avec des tours partant de l'entrepôt et y revenant. Autrement dit, il suffit de s'assurer que si un véhicule part desservir une ou plusieurs villes, alors il doit partir de l'entrepôt, assuré ici par la troisième

contrainte, et il doit aussi y revenir, assuré ici par la quatrième contrainte. La cinquième contrainte s'assure qu'un véhicule effectue bien une tournée, en effet il ne peut desservir la ville j à partir de la ville i, que s'il était déjà en i. Pour la sixième contrainte, on pourrait supposer qu'il n'existe de toute façon pas d'arête allant d'un sommet sur lui même, il n'y aurait donc pas besoin du $l \neq i$. La dernière contrainte est une inégalité MTZ, qui permet d'interdire les sous-tours, mais qui reste polynomial.

On pourra être amené par la suite à diminuer le nombre de véhicules, si la combinatoire du problème devient trop grande.

2) Ici, on considère un problème robuste avec des incertitudes sur les distances. L'ensemble des valeurs que peuvent prendre les durées est :

$$\mathcal{U} = \left\{ \{t'_{ij} = t_{ij} + \delta^1_{ij}(\hat{t_i} + \hat{t_j}) + \delta^2_{ij}\hat{t_i}\hat{t_j}\}_{ij\in A} : \sum_{ij\in A} \delta^1_{ij} \leq T, \sum_{ij\in A} \delta^2_{ij} \leq T^2, \delta^1_{ij} \in [0,1], \delta^2_{ij} \in [0,2] \forall ij \in A \right\} \tag{2}$$

avec $T \in \mathbb{N}$.

On obtient alors la modélisation robuste suivante :

MILP robuste

Objectif :

$$\min_{x_{kij}} \max_{t'_{ij}\in\mathcal{U}} \sum_{(i,j)\in A} \sum_{k=1}^{n} t'_{ij} x_{ijk} \tag{3}$$

Sous contraintes :

$$\begin{aligned}
\sum_{(i,j)\in A} \sum_{k=1}^{n} x_{ijk} &= 1, \quad \forall j \in [\![2;n]\!] \\
\sum_{(i,j)\in A, j\neq 1} d_j x_{ijk} &\leq C, \quad \forall k \in [\![1;n]\!] \\
\sum_{(1,j)\in A} x_{1jk} &\geq \frac{1}{n^2} \sum_{(i,j)\in A, i\neq 1} x_{ijk}, \quad \forall k \in [\![1;n]\!] \\
\sum_{(i,1)\in A} x_{i1k} &\geq \frac{1}{n} \sum_{(1,j)\in A, j\neq 1} x_{1jk}, \quad \forall k \in [\![1;n]\!] \\
x_{ijk} &\leq \sum_{(l,i)\in A, l\neq i} x_{lik}, \quad \forall k,i,j \in [\![1;n]\!] \\
u_{jk} &\geq u_{ik} + 1 - n(1 - x_{ijk}), \quad \forall k, j \in [\![1,n]\!], j \neq 1 \\
x_{ijk} &\in \{0,1\}, \quad \forall (i,j) \in A, \quad k \in [\![1,n]\!] \\
u_{ik} &\in \mathbb{N}, \quad \forall i,k \in [\![1,n]\!]
\end{aligned}$$

3) Résolution par plans coupants et LazyCallback

a) Utilisation d'une variable épigraphique

MILP robuste

Objectif :

$$\min_{z,x_{ijk}} z \tag{4}$$

Sous contraintes :

$$\begin{aligned}
z &\geq \sum_{(i,j)\in A}\sum_{k=1}^{n} t'_{ij}x_{ijk}, \quad \forall (t'_{ij})_{(i,j)\in A} \in \mathcal{U} \\
\sum_{(i,j)\in A}\sum_{k=1}^{n} x_{ijk} &= 1, \quad \forall j \in [\![2;n]\!] \\
\sum_{(i,j)\in A, j\neq 1} d_j x_{ijk} &\leq C, \quad \forall k \in [\![1;n]\!] \\
\sum_{(1,j)\in A} x_{1jk} &\geq \frac{1}{n^2} \sum_{(i,j)\in A, i\neq 1} x_{ijk}, \quad \forall k \in [\![1;n]\!] \\
\sum_{(i,1)\in A} x_{i1k} &\geq \frac{1}{n} \sum_{(1,j)\in A, j\neq 1} x_{1jk}, \quad \forall k \in [\![1;n]\!] \\
x_{ijk} &\leq \sum_{(l,i)\in A, l\neq i} x_{lik}, \quad \forall k,i,j \in [\![1;n]\!] \\
u_{jk} &\geq u_{ik} + 1 - n(1-x_{ijk}), \quad \forall k,j \in [\![1,n]\!], j \neq 1 \\
x_{ijk} &\in \{0,1\}, \quad \forall (i,j) \in A, \quad k \in [\![1,n]\!] \\
u_{ik} &\in \mathbb{N}, \quad \forall i,k \in [\![1,n]\!] \\
z &\geq 0
\end{aligned}$$

b) Soit $\mathcal{U}^*$ l'ensemble utilisé initialement dans le problème maître, ici on construira cet ensemble à travers le programme linéaire suivant :

Objectif :

$$\max_{\delta^1_{ij},\delta^2_{ij}} \sum_{(i,j)\in A} (t_{ij} + \delta^1_{ij}(\hat{t_i} + \hat{t_j}) + \delta^2_{ij}\hat{t_i}\hat{t_j}) \tag{5}$$

Sous contraintes :

$$\begin{aligned}
\sum_{(i,j)\in A}^{n} \delta^1_{ij} &\leq T \\
\sum_{(i,j)\in A}^{n} \delta^2_{ij} &\leq T^2 \\
0 \leq &\delta^1_{ij} \leq 1, \quad \forall (i,j) \in A \\
0 \leq &\delta^2_{ij} \leq 2, \quad \forall (i,j) \in A
\end{aligned}$$

c) Le sous-problème nécessaire à la résolution du problème par plans coupants est le suivant :

Objectif :

$$\max_{\delta^1_{ij},\delta^2_{ij}} \sum_{(i,j)\in A}\sum_{k=1}^{n} (t_{ij} + \delta^1_{ij}(\hat{t_i} + \hat{t_j}) + \delta^2_{ij}\hat{t_i}\hat{t_j})x^*_{ijk} \tag{6}$$

Sous contraintes :

$$\begin{aligned}
&\sum_{(i,j)\in A}^{n} \delta_{ij}^1 \leq T \\
&\sum_{(i,j)\in A}^{n} \delta_{ij}^2 \leq T^2 \\
&0 \leq \delta_{ij}^1 \leq 1, \quad \forall (i,j) \in A \\
&0 \leq \delta_{ij}^2 \leq 2, \quad \forall (i,j) \in A
\end{aligned}$$

d) Pour qu'une solution $(z^*, (x^*_{ijk})_{(i,j)\in A, 1\leq k\leq n})$ du problème maître soit optimale, il faut que :

$$z^* = \max_{(t'_{ij})_{(i,j)\in A}\in \mathcal{U}} \sum_{(i,j)\in A} \sum_{k=1}^{n} t'_{ij} x^*_{ijk} \tag{7}$$

e) Les coupes ajoutées par ce sous-problème sont de la forme :

$$z \geq \sum_{(i,j)\in A} \sum_{k=1}^{n} t^*_{ij} x^*_{ijk} \tag{8}$$

où $(t^*_{ij})_{(i,j)\in A}$ représente la solution optimale en résolvant le sous-problème avec la solution $(x^*_{ijk})_{(i,j)\in A, 1\leq k\leq n}$ du problème maître précédent.

4) Résolution par dualisation
a)

$$\begin{aligned}
&\min_{x_{ijk}} \max_{\delta_{ij}^1, \delta_{ij}^2} \sum_{(i,j)\in A} \sum_{k=1}^{n} (t_{ij} + \delta_{ij}^1(\hat{t_i} + \hat{t_j}) + \delta_{ij}^2 \hat{t_i}\hat{t_j}) x_{ijk} \\
&= \min_{x_{ijk}} \left(\sum_{(i,j)\in A} \sum_{k=1}^{n} x_{ijk} t_{ij} + \max_{\delta_{ij}^1, \delta_{ij}^2} \sum_{(i,j)\in A} \sum_{k=1}^{n} x_{ijk} (\delta_{ij}^1(\hat{t_i} + \hat{t_j}) + \delta_{ij}^2 \hat{t_i}\hat{t_j}) \right)
\end{aligned} \tag{9}$$

b)Le problème interne lié aux variables $\delta_{ij}^1, \delta_{ij}^2$ est le suivant :
Objectif :

$$\max_{\delta_{ij}^1, \delta_{ij}^2} \sum_{(i,j)\in A} \sum_{k=1}^{n} (t_{ij} + \delta_{ij}^1(\hat{t_i} + \hat{t_j}) + \delta_{ij}^2 \hat{t_i}\hat{t_j}) x^*_{ijk} \tag{10}$$

Sous contraintes :

$$\begin{aligned}
&\sum_{(i,j)\in A} \delta_{ij}^1 \leq T \\
&\sum_{(i,j)\in A} \delta_{ij}^2 \leq T^2 \\
&0 \leq \delta_{ij}^1 \leq 1, \quad \forall (i,j) \in A \\
&0 \leq \delta_{ij}^2 \leq 2, \quad \forall (i,j) \in A
\end{aligned}$$

c) Le dual du problème précédent est :

Objectif :

$$\min_{y_{1,T},y_{2,T},y_{1,ij},y_{2,ij}} Ty_{1,T} + T^2 y_{2,T} + \sum_{(i,j)\in A} (y_{1,ij} + 2y_{2,ij}) \tag{11}$$

Sous contraintes :

$$\begin{aligned}
y_{1,T} + y_{1,ij} &\geq (\hat{t_i} + \hat{t_j}) \sum_{k=1}^{n} x_{ijk}, \quad \forall (i,j) \in A \\
y_{2,T} + y_{2,ij} &\geq \hat{t_i}\hat{t_j} \sum_{k=1}^{n} x_{ijk}, \quad \forall (i,j) \in A \\
y_{1,T}, y_{2,T} &\geq 0 \\
y_{1,ij}, y_{2,ij} &\geq 0, \quad \forall (i,j) \in A
\end{aligned}$$

On obtient alors le programme mixte en nombres entiers suivant :
Objectif :

$$\min_{x_{ijk},y_{1,T},y_{2,T},y_{1,ij},y_{2,ij}} \sum_{(i,j)\in A} \sum_{k=1}^{n} x_{ijk} t_{ij} + Ty_{1,T} + T^2 y_{2,T} + \sum_{(i,j)\in A} (y_{1,ij} + 2y_{2,ij}) \tag{12}$$

Sous contraintes :

$$\begin{aligned}
\sum_{(i,j)\in A} \sum_{k=1}^{n} x_{ijk} &= 1, \quad \forall j \in [\![2; n]\!] \\
\sum_{(i,j)\in A, j\neq 1} d_j x_{ijk} &\leq C, \quad \forall k \in [\![1; n]\!] \\
\sum_{(1,j)\in A} x_{1jk} &\geq \frac{1}{n^2} \sum_{(i,j)\in A, i\neq 1} x_{ijk}, \quad \forall k \in [\![1; n]\!] \\
\sum_{(i,1)\in A} x_{i1k} &\geq \frac{1}{n} \sum_{(1,j)\in A, j\neq 1} x_{1jk}, \quad \forall k \in [\![1; n]\!]
\end{aligned}$$

$$\begin{aligned}
x_{ijk} &\leq \sum_{l=1, l\neq i}^{n} x_{lik}, \quad \forall k, i, j \in [\![1; n]\!] \\
u_{jk} &\geq u_{ik} + 1 - n(1 - x_{ijk}), \quad \forall k, j \in [\![1, n]\!], j \neq 1 \\
y_{1,T} + y_{1,ij} &\geq (\hat{t_i} + \hat{t_j}) \sum_{k=1}^{n} x_{ijk}, \quad \forall i, j \in [\![1, n]\!] \\
y_{2,T} + y_{2,ij} &\geq \hat{t_i}\hat{t_j} \sum_{k=1}^{n} x_{ijk}, \quad \forall i, j \in [\![1, n]\!] \\
y_{1,T}, y_{2,T} &\geq 0 \\
y_{1,ij}, y_{2,ij} &\geq 0, \quad \forall i, j \in [\![1, n]\!] \\
x_{ijk} &\in \{0, 1\}, \quad \forall i, j, k \in [\![1, n]\!] \\
u_{ik} &\in \mathbb{N}, \quad \forall i, k \in [\![1, n]\!]
\end{aligned}$$

2 Résolution numérique

Pour cette résolution, la précédente modélisation a été abandonnée au profit de la modélisation transmise en décembre. De fait, l'ancienne modélisation comprenait environ : $\mathcal{O}(n * m)$, où n désigne le nombre de villes qui correspondait au nombre de véhicules dans la modélisation précédentes et m désigne le nombre d'arc du graphe. La nouvelle modélisation propose elle un nombre de variables correspondant à : $\mathcal{O}(m)$, ce qui permet de considérablement réduire le nombre de variables et de contraintes à mesure que le nombre de villes augmente. En effet, cette nouvelle modélisation utilise une variante des inégalités MTZ qui en plus d'éviter les sous-tours permettent de tenir compte du nombre de vaccins livrés par chaque véhicule avant qu'il ne retourne au dépôt, permettant ainsi de respecter la contrainte de capacité des véhicules. Alors qu'avant, on considérait des tournées uniques par véhicule, à savoir un véhicule pouvait effectuer une seule tournée, le nombre de tournées effectuées correspondait alors directement au nombre de véhicules utilisés.

Le langage de programmation qui a été utilisé est C++ avec le solveur CPLEX.

2.1 Résolution du problème par dualisation

On implémente ainsi la résolution du modèle par dualisation présentée dans la modélisation corrigée. On résoudra ce problème en premier pour chaque instance afin d'obtenir une borne inférieure qui pourra être utilisée pour comparer la valeur des solutions obtenues avec les autres méthodes.

2.2 Résolution du problème par plans coupants

Ici, on initialise l'ensemble $\mathcal{U}$, qui représente l'ensemble des coûts possibles et considérés par arcs, avec les coûts statiques. Ensuite, on résout le problème maître qui est l'équivalent du problème statique pour cette première itération, c'est d'ailleurs ainsi que l'on obtient directement la valeur optimale du problème statique. Ensuite avec la solution obtenue avec le problème maître on résout le problème esclave qui est celui de maximisation (associé au pire cas pour ce choix de plan de livraisons), si la valeur optimale du problème maître est strictement supérieure à celle du problème esclave, on ajoute la coupe correspondante comme décrit dans la modélisation précédente et on itère jusqu'à ce que ces deux valeurs soient égales.

Cependant, ici le rajout de la contrainte a été modifié afin d'accélérer la résolution du problème par cette méthode de plans coupants. En effet, le problème réside surtout dans la symétrie des solutions, de fait pour les instances euclidiennes les coûts sur les arcs sont symétriques, il existe alors un nombre important de solutions différentes mais au coût totalement identique, il suffit d'inverser une tournée pour obtenir une solution différente mais de même valeur. Ceci posait problème car suite à l'ajout d'une contrainte du problème esclave comme indiqué dans la modélisation précédente, le problème maître pouvait produire une solution différente de la précédente en inversant tous les arcs, ainsi on obtenait la même valeur que la solution précédente, il fallait donc à nouveau recalculer un problème esclave. Ceci avait pour conséquence de doubler le temps de l'algorithme.

Pour remédier à cela, l'idée a été de dupliquer les δ^1_{ij} et δ^2_{ij} sur les paires d'arcs entrants et sortants. En effet, dans une solution réalisable on ne peut pas avoir à la fois l'arc (u,v) et l'arc(v,u) activés, sauf dans le cas des boucles simples. Dans le cas de la présence d'une boucle simple, à savoir qu'un camion ne déssert qu'une seule ville durant sa tournée, cette méhode de duplication des δ augmente le coût d'une solution optimale en prenant un ensemble de δ qui n'est pas dans $\mathcal{U}$. On recalcule donc à la fin le coût du plan de livraison calculé par le programme maître à l'aide du problème esclave, ce cas très rare est arrivé pour l'instance euclidienne à 9 villes. Deux cas sont possibles, soit la solution obtenue au final est

bien la solution optimale du problème, soit la solution optimale contenait une double boucle mais qui n'a pas été conservée à cause de cette duplication, dans ce cas-là on obtient une solution qui n'est pas optimale. Ceci peut arriver si le fait de dupliquer les δ sur une boucle simple créer un coût tellement élevé, alors que sans la duplication la boucle simple aurait eu un coût beaucoup moins élevé, que cette solution optimale ne l'est plus aux yeux du problème maître.

Cette méthode a aussi été appliquée lors de la résolution des instances non euclidiennes, en effet elle permet de rajouter des coupes réalisables car l'ensemble des δ activés vérifie toujours les conditions de l'ensemble $\mathcal{U}$, hors présence de boucles simples.

Enfin, il se peut que si jamais la limite de temps est atteinte alors le solveur renvoie la dernière solution entière obtenue, or lors de le résolution des précédents problèmes esclaves il se peut qu'il est obtenu une meilleure solution entière, il faut donc veiller à conserver celle-ci et à la renvoyer en cas de limite de temps atteinte, car alors l'optimalité de la solution maître finale n'est plus garantie.

2.3 Résolution du problème par Branch-and-cut

Ici on utilise la classe des Lazy_callback implémentée dans CPLEX, qui permet de rajouter des coupes à chaque solution entière trouvée lors de l'algorithme de branch-and-bound.

Il est tout d'abord important de noter un phénomène qui peut se produire lorsque la limite de temps accordée au solveur est atteinte et donc que le solveur ne renvoie pas une solution optimale. La valeur de la solution renvoyée peut en l'occurence être supérieure à la valeur robuste associée à cette solution, en effet comme la résolution est arrêtée avant l'optimum rien ne garantit que la variable épigraphique z est égale à $\sum t'_{ij}x_{ij}$, elle peut donc lui être strictement supérieure. Néanmoins, on observe que ce n'est pas le cas, la variable épigraphique est toujours strictement inférieure à la valeur du problème esclave associé, hors optimum, la différence vient alors probablement de l'après callback. En effet, une fois la coupe ajoutée, le solveur peut juste augmenter la valeur de la précédente variable épigraphique en conservant toutes les autres valeurs des x_{ij}, ainsi on a bien une solution réalisable. Pour s'assurer que cette augmentation permette en réalité de ne pas dépasser la valeur du problème esclave associée, il suffit d'ajouter en plus de la coupe de robustesse, une coupe qui force z à être inférieur ou égale à la plus petite valeur robuste trouvée jusqu'à présent. Ainsi, on n'ajoute que des coupes valides, avec néanmoins le fait que chaque nouvelle coupe de ce type ajoutée rend inactives les précédentes coupes de ce même type. Par conséquent, la valeur de la solution renvoyée par le solveur correspondra bien à la valeur robuste associée, de plus cette méthode devrait pouvoir accélerer la résolution, avec cependant le défaut qu'il pourrait y avoir des valeurs de variables épigraphiques strictement supérieure à la meilleure solution robuste trouvée jusqu'à présent, qui ne sont donc pas optimales, mais dont la valeur robuste qui leur est associée est strictement inférieure à la précédente. Par conséquent, on nuirait alors à la qualité de la meilleure solution trouvée.

L'idée est alors de conserver à mesure de chaque lazy-callback la plus petite valeur du problème esclave qui a été obtenue avec la solution qui lui est associée.

2.4 Résolution par heuristique

L'heuristique utilisée ici est basée sur trois principes. Le premier est la notion de métaheuristique génétique, travaillant donc avec des populations de solutions, le deuxième est la notion de méthode tabou, qui va être présente lors des générations de nouvelles solutions et le dernier est celui de remplissage

des tournées à partir d'arcs présélectionnés.

Concernant le dernier point, on suppose une liste d'arcs préselectionnés et ordonnées qui vont servir à la création de solution. On considère pour cela trois fonctions :

- **int completing_travelling_backward**(Point first_selected_arcs, vector<vector<Point»& pre_activated_out_arcs, vector<vector<Point»& pre_activated_in_arcs, vector<int> d, int C, vector<bool>& activated_arcs, vector<bool>& visiting_cities, int& total_demand, vector<int> out_depot_arcs) ;
- **int completing_travelling_forward**(Point first_selected_arcs, vector<vector<Point»& out_arcs, vector<vector<Point»& in_arcs, vector<int> d, int C, vector<bool>& activated_arcs, vector<bool>& visiting_cities, int& total_demand, vector<int> in_depot_arcs) ;
- **void completing_total_traveling**(vector<vector<Point»& pre_activated_out_arcs, vector<vector<Point»& pre_activated_in_arcs, vector<Point> Arcs, vector<bool> pre_activated_arcs, vector<int> d, int C, vector<bool>& visiting_cities, vector<bool>& activated_arcs) ;

L'idée est la suivante la première fonction est appelée sur un arc qui est alors activé, puis la fonction va essayer de complèter la tournée contenant cet arc en la remontant par l'arrière tant que la demande total qu'effectue cette tournée ne dépasse pas la capacité du camion et temps qu'il y a des arcs accessibles parmi ceux présélectionnés. Si aucun arc n'est disponible où que la capacité du camion est saturée, alors la fonction rattache la dernière ville ajoutée, qui va ici correspondre à la première ville visitée par le camion, au dépôt. Ensuite, on repart à nouveau du premier arc sélectionné mais cette fois on complète la tournée par l'avant, à nouveau on s'arrête dès que la capacité du camion est atteinte en tenant bien compte de la première moitié de la tournée de cet arc créée par la précédente fonction. Quand on ne peut plus rajouter d'arc, on rattache alors la tournée au dépôt.

A chaque arc ajouté, on enlève les arcs qui étaient soit incidents soit sortant des villes qui ont pu être reliées grâce à une tournée. Ainsi, le nombre d'arcs possibles diminue à chaque ajout d'arcs dans une tournée, une fois tous les arcs épuisés, on regarde si toutes les villes sont joignables par une tournée, sinon on les relie au dépôt à l'aide d'une boucle simple. On obtient alors une solution réalisable du problème dont on peut évaluer le coût robuste en résolvant le sous-problème esclave associé à cette solution.

Concernant à présent la présélection des arcs, l'idée est tout d'abord de créer une première population de solutions, pour cela on crée une toute première solution en présélectionnant tous les arcs, puis en les classant par coût décroissant. En effet, ici on fonctionne avec le pushback des vecteurs dans C++. Ensuite, à partir de la nouvelle solution on va décider de garder une proportion aléatoire, mais néanmoins majorée, d'arcs qui vont être placés tout à droite du vecteur d'arcs présélectionnés. Après, on choisit une proportion d'arcs interdits qui eux ne seront pas présélectionnés, puis on rajoute les arcs restants dans le vecteur d'arcs présélectionnés à nouveau par coût décroissant. On obtient alors une nouvelle solution que l'on rajoute à la population qui désormais comprend deux solutions, et on réitère cette phase à partir de la dernière solution créée jusqu'à avoir atteint la taille de population souhaitée.

Une fois que la première population est créée, il va falloir la faire muter. Pour cela, on sélectionne une partie de la population qui va être conservée comme telle lors de la prochaine génération et une autre qui va être brassée. On sélectionne comme solutions à muter celles ayant le plus bas coût, puis on découpe cette sélection en deux, une solution A va alors recevoir d'une solution B une partie de ses arcs qu'elle considèrera donc comme arcs présélectionnés et placera alors tout à droite de son vecteur d'arcs

de ce type. Ensuite la solution A va pouvoir garder une partie de ses arcs et s'en voir interdir une autre, les proportions des arcs conservés ainsi que leur indices sont choisis aléatoirement, en ayant toutefois majoré les proportions de sélections. On obtient ainsi de nouvelles solutions. On itère alors jusqu'à avoir atteint un nombre de générations fixé.

2.5 Comparaison des différentes méthodes

La méthode par dualisation semble être la plus efficace, c'est la plus rapide, en effet c'est la dernière méthode à atteindre la limite de temps et ainsi renvoyer une solution non-optimale ceci au niveau de l'instance 14-euclidean-true. En effet les méthodes par branch-and-cut et plans coupants avaient déjà atteint la limite de temps pour l'instance 12-euclidean-true, en restant néanmoins proroche de la meilleure borne pour ce qui est de la méthode des plans coupants.

Concernant le type d'instance, on remarque que les instances euclidiennes mettent significativement plus de temps à être résolues. Le fait donc d'avoir un nombre important de solutions équivalentes gène considèrablement la résolution de ce problème d'optimisation. On peut voir que le temps de résolution peut exploser pour certaines instances euclidiennes face à leur instance jumelle non-euclidienne. La méthode la plus touchée semble être celle des plans coupants, malgré le fait de briser les symétries à l'aide de la duplication des δ. Ceci est probablement dû au fait que la présence de symétrie impacte grandement la résolution du problème maître qui doit être résolu un certain nombre de fois avec cette méthode. Néanmoins, le temps limite accordé à la résolution du problème maître pour la méthode des plans coupans est bien plus faible que celui de la méthode par dualisation et branch-and-cut. En effet, comme il faut potentiellement résoudre plusieurs problèmes maîtres et que l'algorithme s'arrête, hors présence d'une limite de temps, que quand la valeur du problème maître et du problèe esclave sont égales, on pourrait se retrouver avec un temps de résolution significativement plus long pour cette méthode si on gardait la même limite de temps. Ainsi, on divise en moyenne par 10 la limite de temps et ajoute une limite de temps globale pour la méthode des plans coupants qui est elle bien la même que pour la méthode par dualisation et branch-and-cut. Ainsi, on résout en moyenne 10 problèmes esclaves puis on s'arrête.

Il est alors intéressant de noter que comme on arrête la résolution du problème maître significativement plus tôt qu'avec les autres méthodes, on peut trouver très tôt au cours de la résolution une solution robuste qui soit plutôt très bonne et d'ailleurs même meilleure que la dernière que renvoierait la méthode après avoir atteint la limite de temps globale. On conserve alors au cours des résolutions du problème esclave la meilleure solution robuste trouvée jusqu'à présent.

Concernant plus particulièrement l'heuristique utilisée, on observe que plus les instances grandissent moins les solutions s'améliorent lors de la génération de populations. En effet, on peut observer que la meilleure solution trouvée a très souvent été générée dans les premières population et donc toutes les populations suivantes n'ont produit aucune meilleure solution. Ainsi dans un premier temps on a rajouté une liste tabou d'arcs pour chaque membre de la population qui était remplie à mesure des générations en étant néanmoins de taille fixée et dont la durée des interdictions de chaque arc une fois rentré dans la liste tabou était en moyenne de 7 générations. Ceci a permis d'améliorer les scores pour les instances de taille moyenne, mais ceci n'a pas suffit pour les très grandes instances. L'idée a donc été dans un second temps de remettre à zéro la population à une génération donnée. Pour ce faire chaque solution de cette population est remplacée par la meilleure solution trouvée jusqu'à présent et ceci est effectuée pour les dix générations suivantes. Cette méthode n'a pas vriment permis d'améliorer les résultats, en effet en vue de cette méhode la proportion de chaque individu gardée a été augmentée, mais les résultats semblent montrer qu'il est plus efficace d'augmenter la variabilité des solutions en rajoutant une liste

tabou et augmentant le nombre d'arcs bannis d'une solution à une autre. En outre, à mesure que la taille des instances augmentait, le temps de création d'une solution par la méthode de complétion devenait de plus en lus long, tant est si bien qu'il a fallu commncer à réduire la tialle des populations et le nombre de générations.

De plus pour ce qui est de la méthode de branch and cut, il semble que ces performances deviennent de plus en plus mauvaise à mesure que la taille des instances grandit. Le fait d'avoir rajouté la contrainte qui indique que la valeur du problème maître doit être inférieure à la dernière plus petite valeur du problème esclave trouvée ne semble pas avoir amélioré. Il serait étonnant que cela soit dû à la présence de contraintes inutiles, car celles-ci sont en faible nombre. En effet, comme expliqué précédemment, sitôt qu'une plus petite valeur de porblème esclave est trouvée, on ajoute une contrainte qui rend inutile toutes les autres contraintes de ce type qui ont été ajoutées avant.

On peut enfin observer que la meilleure méthode semble être celle de la dualisation qui offre quasiment à chaque fois la meilleure solution, bien qu'il est à noter que la méthode par plans coupants ne performant pas particulièrement sur les petites instances fournit de très bons résultats sur les plus grandes, et notamment sur 3 instances c'est elle qui fournit la meilleure solution.

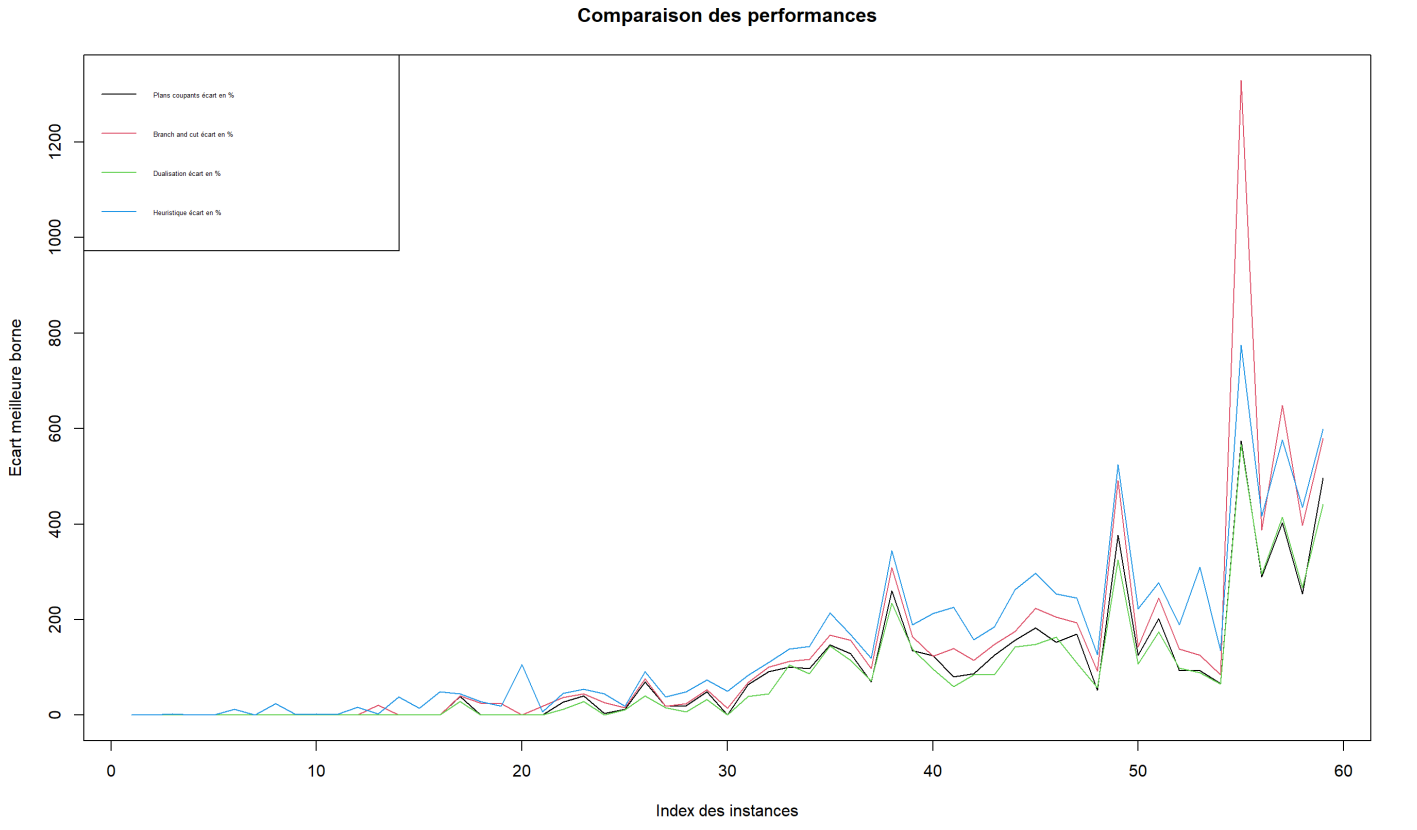

Figure 1 – Comparaison des méthodes avec l'écart en % à la meilleure borne

Ici l'indice des instance correspond à leur ordre de résolution, à savoir les instances de 6 villes à 95, avec pour chaque nombre l'instance euclidienne puis l'instance non-euclidiennes.

On remarque alors effectivement que l'écart à la meilleure borne augmente avec la taille des instances et que la méthode par dualisation est la plus performante en étant toutefois très proche de la méthode de résolution par plans coupants.

	Instance	Valeur statique	PR en %	Meilleure borne	Plans coupants t	Plans coupants é	Branch-and-cut t	Branch-and-cut é	Dualisation temp	Dualisation écar	Heuristique tem	Heuristique écart en %
2	6-euclidean_true	3022.00	2.15	3087.00	0.13	0.00	0.04	0.00	0.04	0.00	0.04	0.00
3	6-euclidean_fals	2554.00	29.48	3307.00	0.17	0.00	0.04	0.00	0.08	0.00	0.24	0.39
4	7-euclidean_true	2983.00	4.99	3132.00	0.56	0.00	0.35	0.00	0.12	0.00	6.43	2.49
5	7-euclidean_fals	1857.00	11.25	2066.00	0.13	0.00	0.02	0.00	0.04	0.00	0.10	0.00
6	8-euclidean_true	3574.00	5.65	3776.00	0.30	0.00	0.26	0.00	0.09	0.00	12.74	0.00
7	8-euclidean_fals	2335.00	40.04	3270.00	0.42	0.00	0.23	0.00	0.09	0.00	24.19	12.20
8	9-euclidean_true	3529.00	18.48	4181.00	1.38	0.00	0.77	0.00	0.16	0.00	0.21	0.12
9	9-euclidean_fals	2699.00	15.34	3113.00	0.29	0.00	0.18	0.00	0.25	0.00	12.97	24.64
10	10-euclidean_tru	4835.00	12.16	5423.00	2.21	0.00	1.28	0.00	0.44	0.00	1.97	1.57
11	10-euclidean_fal	2190.00	22.55	2683.75	1.30	0.01	0.73	0.00	0.27	0.01	12.37	2.24
12	11-euclidean_tru	6222.00	14.79	7142.00	64.43	0.00	9.87	0.00	02.08	0.00	3.56	1.20
13	11-euclidean_fal	2906.00	34.17	3899.00	3.56	0.00	4.69	0.00	0.59	0.00	20.05	16.77
14	12-euclidean_tru	5693.00	11.78	6363.89	323.91	0.33	200.03	20.57	89.24	0.00	43.63	3.00
15	12-euclidean_fal	2946.00	15.04	3389.00	2.90	0.00	1.63	0.00	0.74	0.00	42.15	38.33
16	13-euclidean_tru	4988.00	19.71	5971.00	82.80	0.00	45.86	0.00	4.70	0.00	8.29	14.44
17	13-euclidean_fal	2218.00	20.91	2681.82	2.00	0.01	4.10	0.00	0.97	0.01	35.84	48.85
18	14-euclidean_tru	6569.00	16.73	5525.60	200.53	39.30	200.02	40.06	200.10	27.95	24.62	44.55
19	14-euclidean_fal	3719.00	36.86	5089.69	206.68	0.01	200.03	25.10	25.04	0.01	32.61	28.34
20	15-euclidean_tru	5856.00	4.84	6139.47	715.13	0.01	200.02	24.37	168.41	0.01	28.09	18.56
21	15-euclidean_fal	1854.00	14.66	2125.89	1.31	0.01	3.20	0.00	0.21	0.01	1.33	106.60
22	16-euclidean_tru	8449.00	8.22	9143.32	200.52	0.01	200.06	19.25	25.99	0.01	68.97	6.43
23	16-euclidean_fal	363.00	106.68	750.25	200.64	27.02	200.02	37.16	200.07	11.73	21.59	45.42
24	17-euclidean_tru	6276.00	6.95	4794.63	601.63	39.99	600.02	45.07	600.10	28.57	89.58	53.96
25	17-euclidean_fal	312.00	122.44	694.00	603.65	3.17	600.04	26.77	0.82	0.00	125.78	44.52
26	18-euclidean_tru	8400.00	7.69	9046.26	600.73	12.46	600.01	15.26	600.12	10.91	125.88	18.23
27	18-euclidean_fal	456.00	81.68	828.45	6678.60	69.83	800.05	75.63	800.61	40.27	191.31	90.84
28	19-euclidean_tru	6720.00	3.97	5901.49	120.13	18.39	1000.01	18.83	1000.08	15.54	188.55	37.91
29	19-euclidean_fal	297.00	270.16	1099.39	1022.88	19.98	1000.04	24.52	1000.27	6.99	101.83	48.45
30	20-euclidean_tru	8079.00	14.64	6220.27	400.73	48.88	1000.02	53.63	1003.49	32.84	21.17	73.47
31	20-euclidean_fal	237.00	190.27	687.93	472.23	0.01	1000.05	14.55	321.01	0.01	417.29	50.45

Figure 2 – tableau récaptitulatif pour les instance de 6 à 20 villes

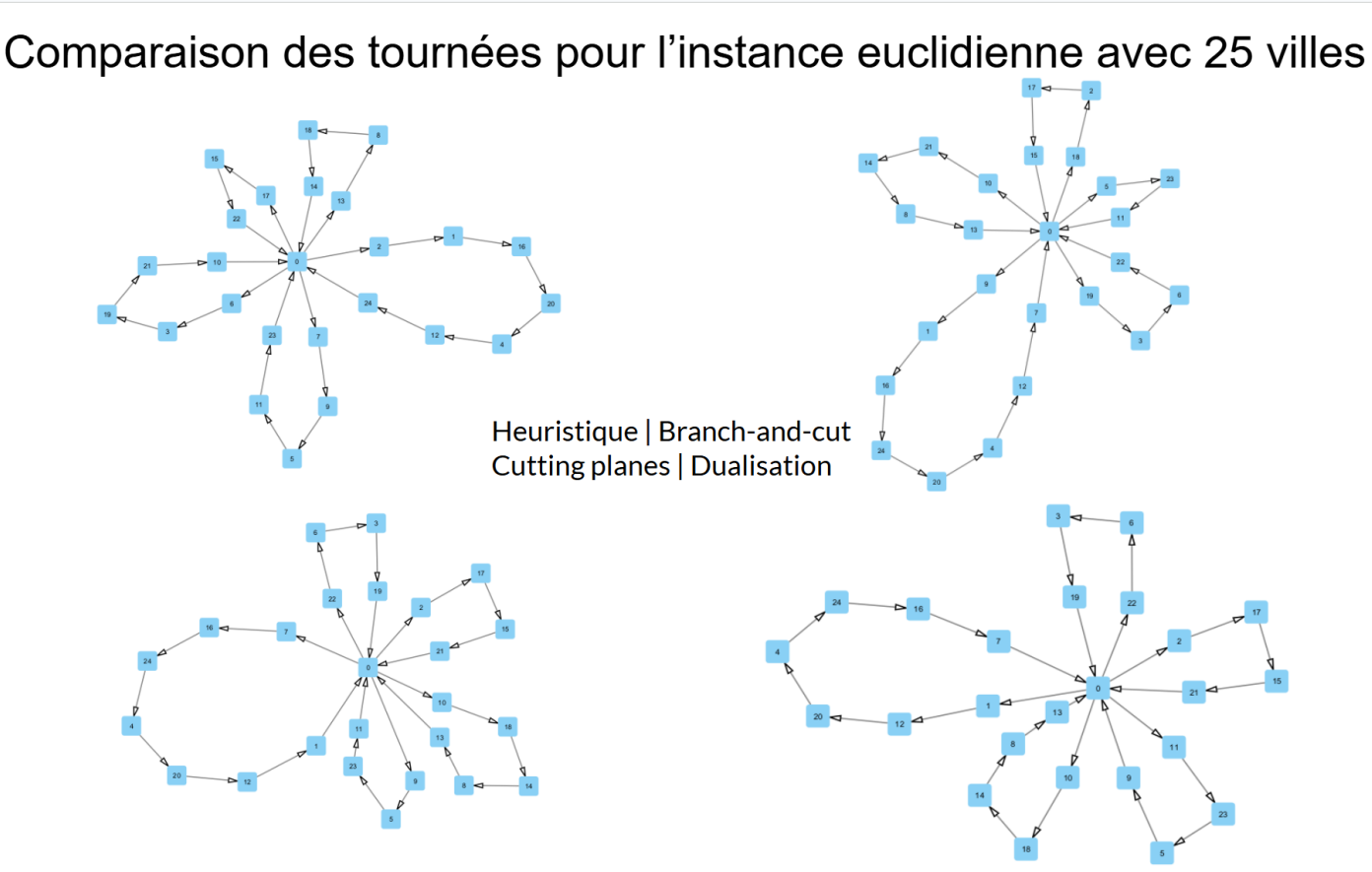

Figure 3 – Comparatif pour 25 villes en version euclidienne

On peut observer que bien que les différentes méthodes aient ici chacune donné une solution différentes, celles-ci se ressemblent assez en termes de structures. En effet, il y a le même nombre de tournées

et les tournées ont des nombres d'arrêts similaires.