# Etude comparative des heuristiques

## S2.02 - Exploration algorithmique d'un problème

Léo Bourdin / Antoine Lindimer / Romain Barabant (Groupe A1)
IUT DIJON AUXERRE  Département informatique

# Table des matières

# Critères de comparaison

## Complexité algorithmique

La complexité algorithmique représente le nombre de calculs basiques que fera l'algorithme selon la taille du problème. Pour un même algorithme, la complexité peut parfois être légèrement différente selon la qualité du code.

**Pour ce critère** : le meilleur algorithme sera celui avec la complexité algorithmique la plus petite.

Nous n'avons malheureusement pas pu calculer exactement les complexités algorithmiques de nos heuristiques mais nous pouvons en revanche les classés par ordre croissant de complexité :

- Plus proche voisin
- Insertion loin et proche
- Recherches locales (voisinage d'une tournée)

## Temps d'exécution

Le temps d'exécution représente la durée que prendra l'algorithme pour calculer l'ordre de la tournée finale. La Stopwatch mesurera et affichera cette durée en milliseconde pour chaque algorithme.

**Pour ce critère** : nous examinerons les temps d'exécution des différents cas du jeu d'essais afin d'en conclure quel algorithme est le plus efficace dans chaque situation.

*Remarque : pour que ce critère soit sensé, on exécutera chaque algorithme sur l'ensemble du jeu d'essai afin qu'il soit soumis à différentes situations.*

*Tous les temps d'exécution sont donnés en millisecondes et ont été calculé sur la même machine pour prendre un même référentiel (elles seront donc surement un peu différentes si les heuristiques sont lancées sur une autre machine).*

## Distance

La distance représente la distance totale de la tournée finale trouvée par l'algorithme.

**Pour ce critère** : nous examinerons les temps d'exécution des différents cas du jeu d'essais afin d'en conclure quel algorithme est le plus efficace dans chaque situation.

*Remarque : pour que ce critère soit sensé, on exécutera chaque algorithme sur l'ensemble du jeu d'essai afin qu'il soit soumis à différentes situations.*

# Présentation des heuristiques

## Plus proche voisin

Nous avons implémenté cette heuristique comme vu lors du TP1.

On commence par l'usine, puis on va au sommet le plus proche non visité grâce à FloydWarshall.

Une fois tous les sommets visités, on retourne à l'usine par le chemin le plus court.

## Insertion (proche et loin)

Nous avons implémenté cette heuristique comme vu lors du TP1.

On commence par déterminer les 2 sommets les plus éloignés que l'on insert dans la tournée, puis, grâce à FloydWarshall et à la méthode de calcul de distance entre un sommet et la tournée actuelle, on cherche le sommet non visité le plus proche ou le plus loin (selon l'heuristique) de la tournée et on l'insert dans cette tournée.

## Voisinage d'une tournée (insertion proche et loin)

Lance une insertion proche ou loin (selon l'heuristique) pour le graphe, échange deux voisins, puis compare la distance de la nouvelle tournée avec l'ancienne, et la remplace si cette première est inférieure, et ainsi de suite jusqu'à avoir échangé les voisins de toute la liste. On répète ceci tant que l'on trouve mieux et que l'on n'échange pas toute la liste plus de 999 fois.

## Recherche aléatoire

Même principe que pour le voisinage d'une tournée, mais au lieu de lancer une insertion, on prend une tournée dans un ordre complètement aléatoire. On répète le processus d'échange des voisins 10 000 fois.

*Remarque : vous trouverez dans le fichier Heuristique, le code (bien sûr commenté) des heuristiques implémentées.*

# Jeu d’essai de graphes

Comme susdit, le **jeu d’essai** aura pour but de **faire rencontrer aux heuristiques des situations diverses et variées** afin de récolter des données plus précises, donc utiles pour la comparaison. Nous allons donc **explorer différents cas concrets** de la vie utilisant la notion de graphes.

Pour créer un jeu d’essai pertinent, nous avons commencé par recréer les graphes que nous avions étudié en séance de TP. Puis nous avons décidé de créer nos propres graphes afin d’obtenir des résultats exploitables.

Avec des graphes simples (où l’on peut obtenir facilement un résultat en faisant tourner les algorithmes à la main), les résultats ne sont pas très intéressants. Le temps d’exécution est inférieur à 10 ms dans tous les cas, on observe tout de même de petites différences dans les résultats des tournées. Ces résultats étaient à prévoir puisque nous les avions observés durant les séances de découvertes.

En utilisant le graphe fourni avec le sujet (GrapheSimple2.gph), nous avons pu tester nos algorithmes et observer des résultats différents en fonction de l’algorithme.

Pour obtenir des résultats intéressants nous avons commencé par créer des graphes où tous les sommets sont reliés entre eux, ce sont des cliques. Nous avons essayé de créer une clique de degré 5 « à la main », cela s’est révélé très long et inutile car il semble impossible de créer des cliques de degré supérieur à 10 « à la main ». Nous avons donc créé un algorithme qui génère automatiquement dans la console les lignes qui permettent de créer un graphe avec un fichier en .gph dans le projet.

Cet algorithme imprime dans la console :

- L’usine qui est numérotée 0 et ses coordonnées
- La liste des magasins et leurs coordonnées
- La liste des routes entre tous les sommets

Pour les sommets, les coordonnées sont générées en se basant sur le cercle trigonométrique. On place dans un premier temps l’usine à un abscisse défini et avec une ordonnée de 0, puis on place les sommets sur le cercle à intervalle régulier. En ce qui concerne les routes, avec deux boucles imbriquées, on peut générer chaque route entre tous les sommets. Pour ce qui est de la pondération, elle est aléatoire et comprise entre 1 et le nombres de sommets.

Grâce à cet algorithme, nous avons réalisés une clique de taille 10 et une clique de taille 50.

Avec ces deux graphes, nous avons pu tester nos algorithmes et observer des différences en termes de temps d’exécution. Puisque tous les sommets

sont reliés entre eux, on retrouve beaucoup moins de contraintes pour passer d'un sommet à un autre. Ce qui fait que le temps d'exécution est très court pour tous les algorithmes implémentés, aux alentours de 50 ms (des résultats plus détaillés se trouvent dans la partie suivante).

Nous avons ensuite réutilisé le principe de ce programme pour créer une autre catégorie de graphes : des graphes bipartis, plus précisément des graphes bi partie complets.

Définition de graphe biparti :

- Un graphe est dit **biparti** si son ensemble de sommets peut être divisé en deux sous-ensembles disjoints ***U*** et ***V*** tels que chaque arête ait une extrémité dans ***U*** et l'autre dans ***V***. *(Source : Wikipédia)*

Définition d'un graphe biparti complet :

- Un graphe est dit **biparti complet** s'il est **biparti** et que chaque sommet du premier ensemble est relié à tous les sommets du second ensemble. *(Source : Wikipédia)*

Nous avons choisi ce cas particulier pour le fait qu'il était simple d'en générer de grande taille, comme pour les cliques, avec le même principe qu'évoqué précédemment.

Nous avons donc généré deux graphes, un graphe avec 50 sommets et un avec sommets. Pour ce qui est de la pondération des routes, on retrouve le même principe que pour les cliques, la pondération est aléatoire et comprises entre 1 et le nombre de sommets.

Il nous fallait aussi quelques cas particuliers et après quelques recherches, nous avons trouvé deux graphes avec des propriétés particulières.

Dans un premier temps, nous avons recréé le graphe de Petersen (illustration ci-dessous). Ce graphe est composé de 10 sommets et 15 arêtes et il sert habituellement d'exemple et de contre-exemple pour plusieurs problèmes de la théorie des graphes. Pour ce qui s'agit de la pondération des routes. Nous avons créé deux graphes, un ayant toutes ces arêtes de poids 10 et un autre avec une pondération aléatoire.

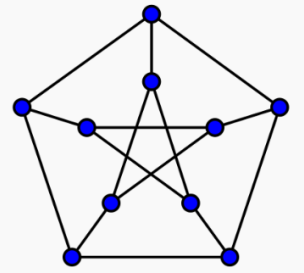

*Figure 1 - (Source : Wikipédia)*

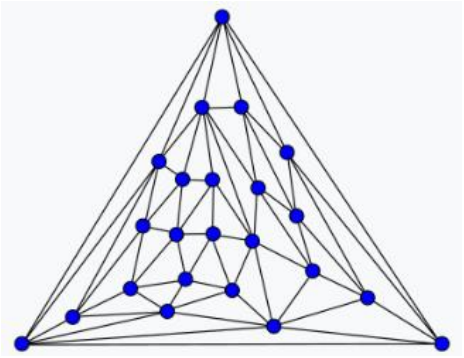

*Figure 2 - (Source : Wikipédia)*

Ensuite nous avons recréé le graphe de Kittel (illustration ci-dessous). Ce graphe est composé de 23 sommets et 63 arêtes, sa configuration est particulière et fait que l'on retrouve 42 faces triangulaires. De la même manière que pour le graphe précédent, nous avons réalisé deux graphes. Un ayant toutes ces arêtes de poids 10 et un autre avec une pondération aléatoire.

Avec ces deux cas particuliers, nous voulions voir si certains algorithmes peu performants pouvait avoir de meilleurs résultats face à ces situations. (Les résultats se trouvent dans la partie suivante)

Idée des graphes trois chemins :

L'idée est simple : laisser le choix à l'heuristique entre 3 chemins interconnectés seulement à leur début et à leur fin. Dans chacun de ces chemins, les sommets sont répartis de manière aléatoire, afin de permettre une multitude de parcours au sein de ceux-ci.

# Mesure qualitative des algorithmes

## Graphes bipartis

Grâce à ces deux graphes, nous avons pu tester nos algorithmes et observer des différences en termes de temps d'exécution. Puisque tous les sommets sont reliés entre eux, on retrouve beaucoup moins de contraintes pour passer d'un sommet à un autre. Ce qui fait que le temps d'exécution est très court pour tous les algorithmes implémentés, aux alentours de 50 ms.

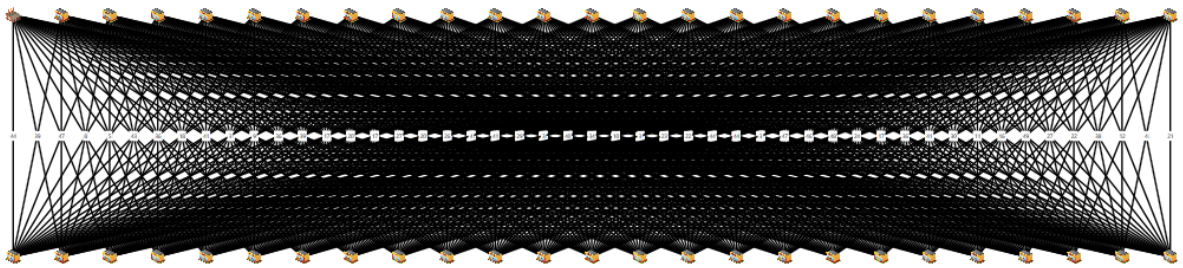

*Figure 3 – Graphe Biparti de 50 sommets*

## Tableau de données

| | Biparti de 50 | | Biparti de 100 | |
|---|---|---|---|---|
| | Temps d'exécution | Distance | Temps d'exécution | Distance |
| Plus proche voisin | **~50** | 268 | **~330** | **508** |
| Insertion proche | ~60 | 243 | ~440 | 524 |
| Insertion loin | ~60 | **241** | ~440 | 530 |
| Recherche locale (insertion proche) | ~450 | 243 | ~1850 | 524 |
| Recherche locale (insertion loin) | ~450 | 241 | ~1850 | 530 |
| Aléatoire | ~280 | ~450 | ~930 | ~1250 |

## Comparaison et analyse

Avec deux tailles de graphes différentes, on obtient des résultats très différents.

Ainsi, plusieurs données ressortent :

**Graphe à Biparti de 50**

- Les insertions et les recherches locales trouvent une distance bien presque identique.
- L'ordre de grandeur des temps d'exécution est le même (de 50 à 60 ms en moyenne)

**Graphe à Biparti de 100**

- Le Plus Proche Voisin a un temps d'exécution ET une distance inférieure à toutes les autres heuristiques.

Pour le graphe biparti de 50, les insertions (proche ou loin) sont les plus adaptées à ce graphe : la différence de distance est négligeable et le temps d'exécution est le même.

Cependant, pour le graphe biparti de 100, l'heuristique du Plus Proche Voisin est bien meilleur que toutes les autres, tant sur la distance que sur le temps d'exécution (440ms contre 330ms : bien plus rapide).

## Graphes cliques

Grâce à ces deux graphes, nous avons pu tester nos algorithmes et observer des différences en termes de temps d'exécution. Puisque tous les sommets sont reliés entre eux, on retrouve beaucoup moins de contraintes pour passer d'un sommet à un autre. Ce qui fait que le temps d'exécution est très court pour tous les algorithmes implémentés, aux alentours de 50 ms.

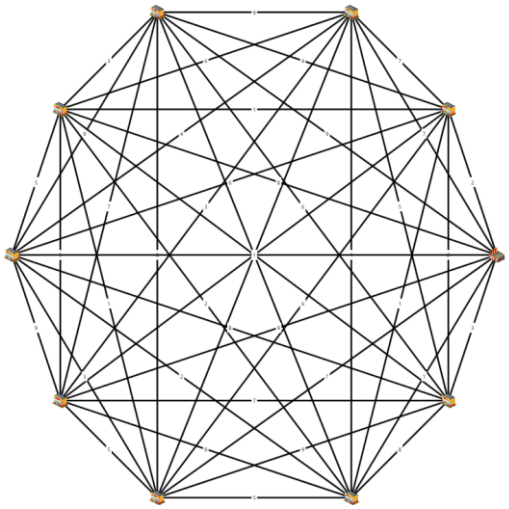

*Figure 4 - Clique de 10*

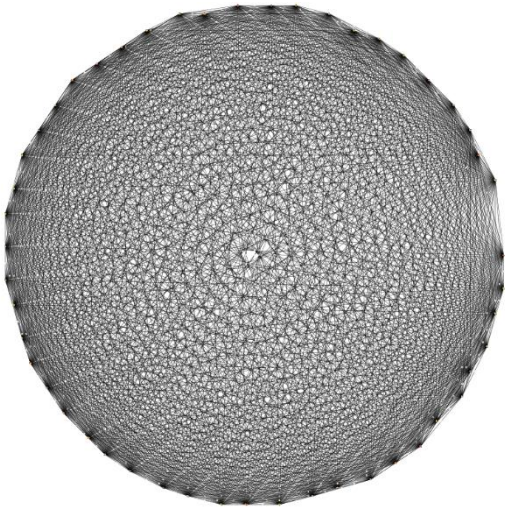

*Figure 5 - Clique de 50*

### Tableau de données

| | Clique de 10 | | Clique de 50 | |
|---|---|---|---|---|
| | Temps d'exécution | Distance | Temps d'exécution | Distance |
| Plus proche voisin | <0ms | 25 | **~50** | 146 |
| Insertion proche | <0ms | **21** | ~60 | 132 |
| Insertion loin | <0ms | 22 | ~60 | **131** |
| Recherche locale (insertion proche) | ~20 | **21** | ~400 | 132 |
| Recherche locale (insertion loin) | ~20 | 22 | ~400 | **131** |
| Aléatoire | ~50 | ~30 | ~280 | ~280 |

## Comparaison et analyse

Avec deux tailles de graphes différentes, on obtient des résultats plutôt similaires.

Ainsi, plusieurs données ressortent :

**Graphe à Clique de 10**

- L'ordre de grandeur des distances des différentes heuristiques est le même du fait de la petite taille du graphe (entre 20 et 30)
- Le temps d'exécution est inférieur à 0ms pour chaque heuristique

**Graphe à Clique de 50**

- Les insertions et les recherches locales trouvent une distance presque identique.
- L'insertion loin trouve une distance de 1 de moins que la recherche locale et l'insertion proche.

Pour ces graphes, aucune heuristique n'est vraiment meilleure que les autres. L'algorithme qui s'en sort le mieux peut être différent selon la taille de la clique et la valeur des arêtes.

Dans leur globalité, les insertions (proche ou loin) sont les plus adaptée à ces graphes, car elles trouvent les meilleures distances avec un temps d'exécution très satisfaisant. Même si nous voulons prioriser la rapidité d'exécution, Le Plus Proche Voisin n'est pas un très bon compromis.

## Graphes Petersen

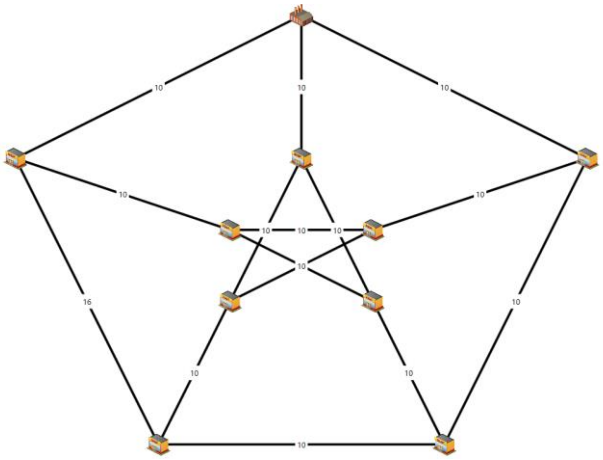

*Figure 6 - graphe de Petersen avec chemins à longueurs constantes*

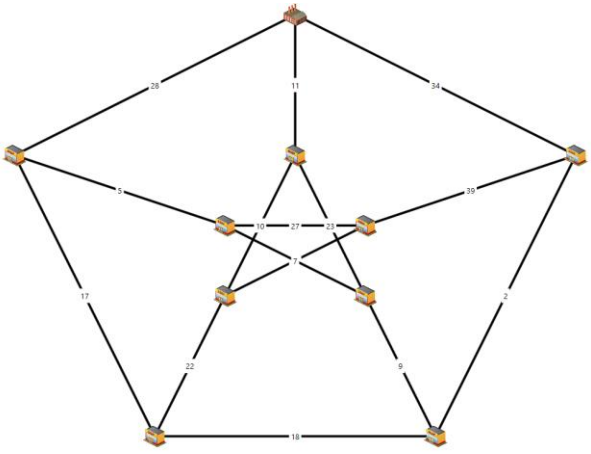

*Figure 7 - graph de Petersen avec chemin à longueur aléatoire*

Comme décrit précédemment, le graphe de Petersen est un graphe particulier possédant 10 sommets et 15 arêtes.

## Tableau de données

| | Arêtes constantes | | Arêtes aléatoires | |
|---|---|---|---|---|
| | Temps d'exécution | Distance | Temps d'exécution | Distance |
| Plus proche voisin | ~0ms | 110 | ~0ms | 154 |
| Insertion proche | ~0ms | 116 | ~0ms | 146 |
| Insertion loin | ~0ms | 110 | ~0ms | 144 |
| Recherche locale (insertion Proche) | ~10ms | 116 | ~15ms | 146 |
| Recherche locale (insertion loin) | ~10ms | 110 | ~15ms | 144 |
| Aléatoire | ~50ms | ~160 | ~50ms | ~210 |

## Comparaison et analyse

Puisque le graphe de Peterson est de petite taille, les temps d'exécution sont extrêmement courts. Pour autant on peut remarquer qu'un algorithme se démarque.

**Graphe à longueurs constantes**

- L'ordre de grandeur des longueurs des tournées est le même pour toutes les heuristiques.
- On peut tout de même voir que l'insertion loin se démarque avec des tournées plus courtes

**Graphe à longueurs aléatoires**

- Même chose que pour le graph avec longueur constante.

On peut en déduire que pour le graphe Peterson, l'algorithme le plus adapté est l'insertion loin. Pour autant, puisque que c'est un graphe avec peu de sommets et d'arêtes on ne peut pas observer de vraies différences dans les temps d'exécution et dans la longueur des tournées.

## Graphe de Kittel

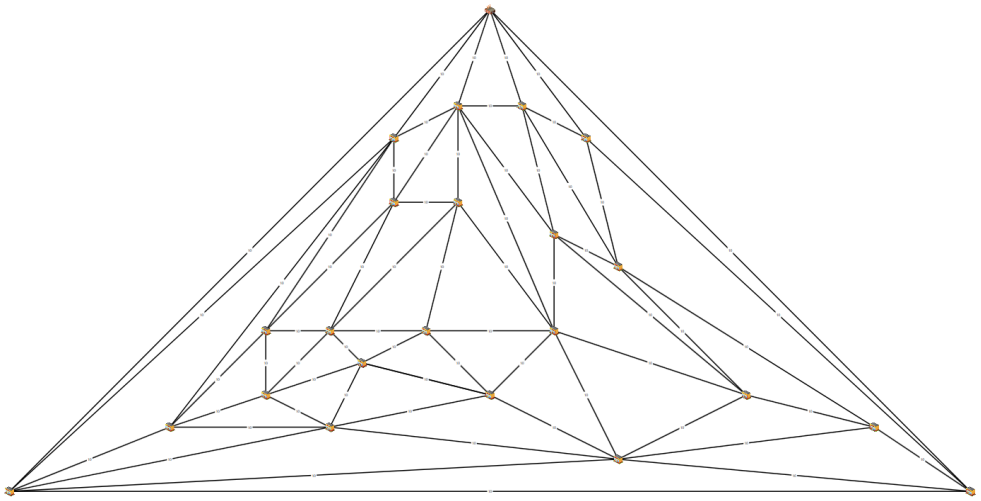

*Figure 8 - Graphe de Kittel*

Comme décrit précédemment, le graphe de Kittel est un graph particulier possédant 23 sommets et 63 arêtes.

### Tableau de données

| | Arêtes constantes | | Arêtes aléatoires | |
|---|---|---|---|---|
| | Temps d'exécution | Distance | Temps d'exécution | Distance |
| Plus proche voisin | ~0ms | 250 | ~0ms | 165 |
| Insertion proche | ~0ms | 250 | ~0ms | 163 |
| Insertion loin | ~0ms | 240 | ~0ms | 170 |
| Recherche locale (insertion Proche) | ~50ms | 250 | ~60ms | 163 |
| Recherche locale (insertion loin) | ~50ms | 240 | ~60ms | 170 |
| Aléatoire | ~75ms | ~460 | ~75ms | ~360 |

### Comparaison et analyse

De même que pour le graph de Peterson, ce graphe est de petite taille donc les temps d'exécution sont très courts.

**Graphe à longueurs constantes**

- L'ordre de grandeur des longueurs des tournées est le même pour toutes les heuristiques.
- On peut tout de même voir que l'insertion loin se démarque avec des tournées plus courtes

**Graphe à longueurs aléatoires**

- Même chose que pour le graphe avec longueur constante.

On peut en déduire que pour le graphe Kittel, l'algorithme le plus adapté est l'insertion loin comme proche. Pour autant, puisque que c'est un graphe avec peu de sommets et d'arêtes on ne peut pas observer de vraies différences dans les temps d'exécution et dans la longueur des tournées.

## Graphes trois chemins

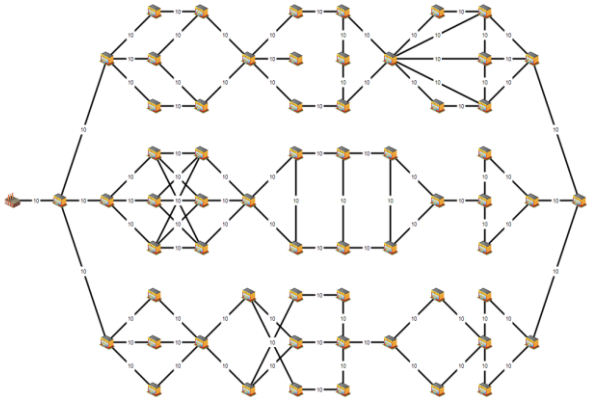

*Figure 9 - Graphe avec chemins à longueurs constantes*

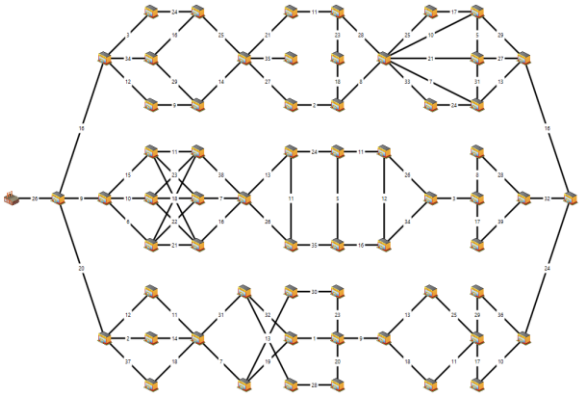

*Figure 10 - Graphe avec chemins à longueur aléatoires*

Pour ces graphes, nous avons voulu observer le comportement de chaque heuristique en lui laissant le choix en trois grands chemins (l'usine étant située tout à gauche).

Nous avons créé deux versions à partir de cette idée : la seule différence est la valeur des arêtes.

La Figure 1 représente le graphe avec les arêtes constantes (10 partout) et la Figure 2 représente le graphe avec les arêtes aléatoires (entre 0 et 40).

## Tableau de données

| | Arêtes constantes | | Arêtes aléatoires | |
|---|---|---|---|---|
| | Temps d'exécution | Distance | Temps d'exécution | Distance |
| Plus proche voisin | **~75** | 900 | **~75** | 1520 |
| Insertion proche | ~100 | **810** | ~100 | 1557 |
| Insertion loin | ~100 | 860 | ~100 | **1279** |
| Recherche locale (insertion proche) | ~500 | **810** | ~150 | 1543 |
| Recherche locale (insertion loin) | ~500 | 860 | ~750 | **1279** |
| Aléatoire | ~80 | ~4000 | ~80 | 5500-6500 |

## Comparaison et analyse

Avec des arêtes aléatoires, on obtient des résultats complètement différents qu'avec des longueurs constantes.

Ainsi, plusieurs données ressortent :

**Graphe à longueurs constantes**

- L'ordre de grandeur des distances des différentes heuristiques est le même (entre 800 et 900, excepté pour l'algo aléatoire)
- Le Plus Proche Voisin à un temps d'exécution plus faible que tous les autres

**Graphe à longueurs aléatoires**

- L'insertion loin trouve une distance bien inférieure à tous les autres
- Le voisinage d'une tournée basé sur l'insertion proche trouve une distance de 14 de moins que l'insertion proche.
- Le Plus Proche Voisin a un temps d'exécution plus faible que tous les autres ET trouve une distance inférieure à l'insertion proche

Pour ces graphes, on peut donc clairement affirmer que selon les arêtes, l'algorithme qui s'en sort le mieux n'est jamais le même.

L'insertion proche est plus adaptée au graphe constant et l'insertion loin au graphe aléatoire. Cependant, nous remarquons que si l'on veut prioriser la rapidité d'exécution, Le Plus Proche Voisin est le meilleur compromis.

# Conclusion générale

A travers les différents graphes nous avons observé l'utilité de chaque algorithme.

**Avec nos exemples**, nous avons conclu plusieurs choses :

- Nous n'avons pas de trouvé de cas où l'**heuristique aléatoire** est adapté
- **Le voisinage d'une tournée (recherche locale)** basé sur l'insertion loin **n'a jamais été plus efficace** dans les cas que nous avons traité (alors que celui basé sur l'insertion proche si)
- **Les recherches locales** s'exécutent, en moyenne, sur une durée supérieure à toutes les autres heuristiques**.**
- Pour les **graphes de Kittel et Petersen**, même s'ils ont des propriétés mathématiques intéressantes, pour le problème du voyageur de commerce, les résultats des différentes heuristiques sont difficilement exploitables.

D'une manière générale, les cas que nous avons étudiés répondent à différents besoins et ont chacun produit des données intéressantes (même si parfois peu exploitable). Ainsi, nous avons comparé la globalité de nos heuristiques implémentées pour réaliser cette étude comparative.