M1 ALMA
Université de Nantes
2010-2011

Projet de TP n°2
Structures complexes et algorithmes

MARGUERITE Alain
RINCE Romain

Université de Nantes
2 rue de la Houssinière, BP92208, F-44322 Nantes cedex 03, FRANCE

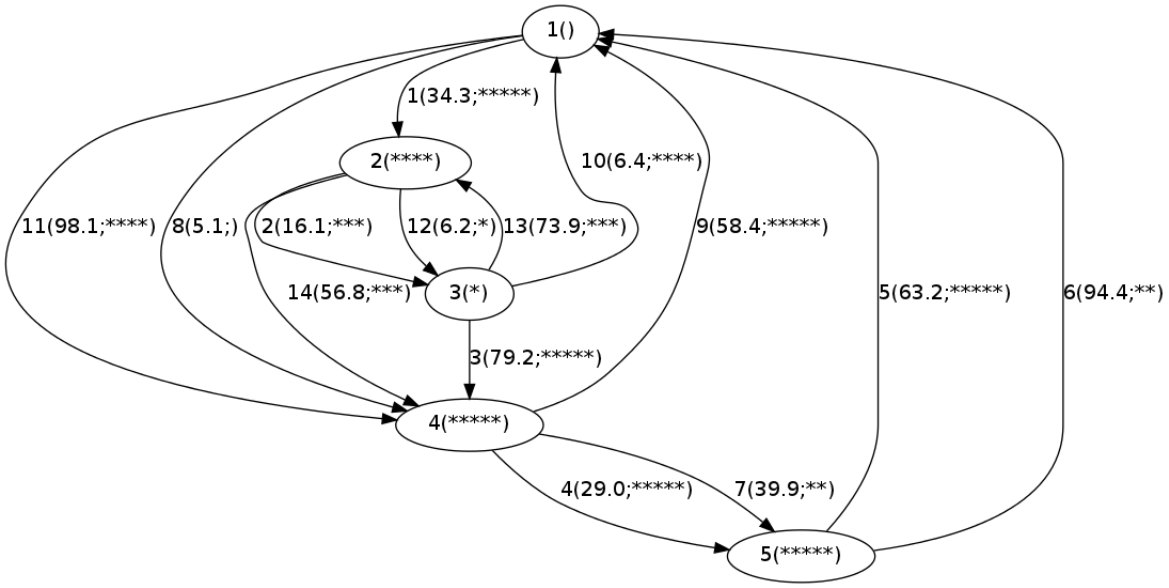
Encadrant : Christophe JERMANN

Table des matières

1 Introduction

1.1 Problème à résoudre

L'objectif de ce projet est de réaliser un Global Positioning System «amélioré». Sans entrer dans les détails, un GPS est réalisé à partir d'une modelisation sous forme de graphe orienté d'une carte où les posistions sont des noeuds et les routes sont des arcs. L'application d'algorithmes de plus courts chemins sur ces graphes permettent alors de répondre au problème concrét du calcul d'iténéraires. Cependant il est rare, dans la vie réelle, que ce type de problème admette une unique contrainte. Ici nous allons travailler sur des graphes plus complexes (voir figure ci-après) par leurs nombres de paramètres



L'idée de réaliser un GPS prenant aussi en compte l'aspect touristique d'un lieu ou d'une route donne les opportunités suivantes :

- Une étude approfondie des algorithmes sur les graphes étudiés en cours pour choisir le plus adapté en fonction du problème et de la structure de donnée concrète.

- Des manipulations plus complexes de ces algorithmes puisqu'ils prennent plusieurs (et non un seul) paramètres en compte.
- Ces différents choix de combinaisons d'implémentations et d'algorithmes entraînes des calculs variés.

Tous ces aspects sont au coeur même du module Structures complexes et algorithmique. La première partie du projet se contentait d'étudier et d'implémenter des structures et des algorithmes complexes. Au terme de l'ultime partie, nous réalisons à partir de ces outils théoriques une application concrète.

1.2 Application réalisée

1.2.1 Structure du programme

Dans la continuité de la première partie du projet, nous avons conçu de nouvelles classes en nous appuyant sur celles de notre graphe orienté. Voici la liste des différents fichiers du package GPS :

- Ville.java : Composée d'un nom et d'une entier qualilté dénombrant le nombre d'étoiles
- Route.java : Héritant de la classe Arc.java, posède un plus une longueur et d'une qualité tout comme Ville.java
- GPS.java : Comporte les différentes méthodes demandées dans le sujet. Le constructeur prend en compte les données fournies par le parseur ainsi que celle saisie sur l'entrée standards par l'utilisateur
- Parser.java : Parse le fichier fourni en paramètre et retourne à la classe appelante le graphe créé ainsi que la distance max entre deux villes sur le graphe, la meilleure qualité, et un annuaire inversé qui donne l'id d'un nœud ou d'un arc dans le graphe à partir de son nom.

1.2.2 Déroulement de l'application

Le programme se lance sans paramètres. Il demande à l'utilisateur d'entrer successivement :

- Le nom du fichier d'entrée (devant être présent dans le dossier courant).
- Le choix d'implémentation (l la liste d'adjacence et m pour la matrice d'adjacence).
- Le nom de la ville de départ.
- Le nom de la ville d'arrivée.
- La valeur (K ou A) du paramètre en fonction de la méthode choisie.

L'itinéraire calculé sera alors affiché selon les consignes du sujet. Le programme se termine par la suite. Il faut relancer le programme un autre calcul ?

2 Algorithmes mis en oeuvre

2.1 Algorithme de plus court chemin

Les deux methodes demandées pour le calul d'itinéaire (par agrégation et à détour borné), il est necessaire d'employer un algorithme de plus court chemin. Notre choix de d'algorithme s'est orienté vers celui Bellman-Ford. Son avantage par rapport à celui de Dijkstra est qu'il est capable de détecter un circuit absorbant. Il peut de la sorte informer l'utilisateur l'echec de a recherche d'un itinéraire compromis selon ces paramètres.

2.2 Methode Agrégation

L'objectif de cette methode est d'obtenir un plus court chemin avec une pondération particulière pour routes calculée au préalable. Il suffit donc d'appliquer un algorithme standard de parcours de graphe en fournissant toutes les pondérations de chaques routes et villes que l'on aura calculé au préalable. N Nous avons procédé de cette manière. Ainsi la fonction $get_agregat(Routeroute, doubleA)$ permet de donner l'agrégat d'une route. Elle est utilisée par $publicArrayList < Double > agregation(doubleA)$ qui retourne un tableau de ces pondération indicé par l'id des routes. Ce tableau est ensuite fourni en paramètre à l'Algorithme de Bellman Ford. Celui ci pourra alors procéder au relâchement des arcs en fonctions de ces pondérations. Dans le cas ou une route améliorante est trouvé, celle ci est stockée dans une hash map à la clef de valeur id de la ville destination de cette route. A la fin de l'algorithme de Bellman-Ford nous avons notre plus court chemin par rapport aux agrégations pré calculées dans cette hash map. Les opérations qui suivent servent uniquement à stocker ce chemin dans le bon sens dans une ArrayList. L'opération d'affichage a en effet du resultat sous cette forme pour le traiter.

2.3 Détour Borné

L'algorithme du détour borné consiste à trouver le plus court chemin entre deux points puis obtenir la longueur de ce chemin. L'algorithme implémenté utilise donc la méthode de Bellman-Ford déjà implémentée pour trouver le plus court chemins. Cependant étant donné que la pondération sur les arcs correspond au distance entre les villes, on peut affirmer qu'il n'y aura pas de circuit absorbant ; dans ce cas Dijkstra aurait permis d'avoir une meilleure complexité pour la recherche du PCC. Une fois le PCC trouvé, l'algorithme va parcourir presque tous les chemins possibles à partir du point d'origine et retourné une chemin borné par la longueur du PCC et un K >= 1. Pour l'implémenter nous avons utiliser une méthode itérative sur chaque arc. Ainsi lorsque l'on arrive sur un nœud le programme s'exécute comme suit :

- Il vérifie s'il a atteint le nœud d'arrivée. Si oui il regarde s'il a une meilleure qualité que celui déjà stocké et le stocke le cas échéant.
- Le choix d'implémentation (l la liste d'adjacence et m pour la matrice d'adjacence).
- Si le nœud n'est pas celui d'arrivée, il cherche tous les arcs sortants dont la longueur ajoutée à la longueur déjà parcouru ne dépasse pas la borne fixée et vérifie que le nœud d'arrivée n'a pas été encore parcouru. Le algorithme s'appelle sur chaque arc valide en grisant au préalable le nœud courant et en ajoutant l'arc au chemin parcouru (implémenté en pile). Une fois que la méthode sur les arcs se terminent toutes, l'algorithme remet à blanc le nœud courant et ôte l'arc du chemin parcouru.
- Le nom de la ville d'arrivée.

3 Analyse théorique

3.1 Etudes de cas

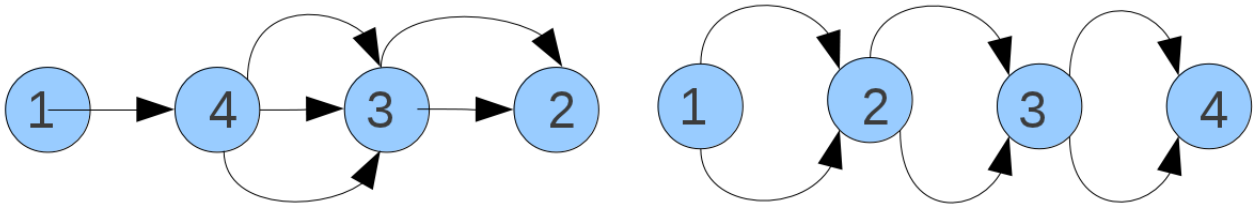
3.1.1 Cas de la méthode d'agrégation

Cette methode repose principalement sur l'algorithme de Bellman-Ford de compléxité : O(|N|*|A|) où |N| est le nombre de ville et |A| le nombre de routes. Il faut aussi fournir le tableau des agrégations. Cette opération consite à opérer pour chaque route la formule et l'ajouter dans un tableau. Sa compléxité est donc en O(|A|). La methode se termine par quelques manipulations pour fournir le plus court chemin dans une ArrayList (cf 2.2). Les deux parcours de ces opérations sont en O(|R|) taille maximun du plus court chemin.
Nous avons donc au total une compléxité de O(|A|*|N|).

Le choix de l'implémentation du graph est choisie selon la performance de *listeArcs*. Sa compléxité est de O(|N|+|A|) pour la liste d'adjacence et en ($O|N|^2$) pour la matrice d'adjacence. Le choix de liste d'adjacence est donc à privilégier.

3.1.2 Cas de la méthode par détour borné

Le détour borné utilise l'algorithme de Bellman-Ford donc nous avons au minimum une complexité O(A*N). Obtenir la longueur du PCC sera dans le pire des cas O(A+N). Il nous faut maintenant calculer le coût de la recherche du meilleur chemin borné. Étant donné que l'algorithme recherche quasiment tous les chemins à partir d'un point, la complexité exprimable en fonction du nombre d'arcs et de nœuds dépend essentiellement de la forme du graphe. Observons l'exemple suivant pour 4 nœuds et 6 arcs et on désire aller de 1 à 4. Dans un cas on peut voir que le nombre de chemins est seulement de 1 pour le premier graphe car l'algorithme ne trouve qu'un chemin à partir de 1 et ne cherche pas les chemins existants vers tous nœuds (bien qu'il y en ait en fait 6). Tandis que pour le second parcours l'on obtient 8 chemins à parcourir. Je ne me sens pas personnellement capable de donner la plus grande quantité de chemins pour un graphe étant donné le nombre de nœuds et le nombre d'arc, je vais donc cherché un cas qui entraîne un fort nombre de chemins. Un



cas simple à étudier est un cas similaire au second exemple ;c'est-à-dire un graphe ou chaque nœuds a tous ses arcs qui pointe vers le nœud « suivant ». Donc si $N < 2A$ on peut faire que chaque nœud a en moyenne $\frac{A}{N-1}$ arcs sortants soit $(\frac{A}{N-1})^{N-1}$ chemins, ou si $N > 2A$ $2^{\frac{A}{2}}$ chemins en répartissant correctement les arcs. Il faut de plus ajouter à cela la mise à jour du meilleur chemin stocké dans une liste qui dans le pire des cas est faite à chaque fois. Le nombre de chemins est donc un paramètre exponentiel mais qui pourra énormément varié selon la forme du graphe, la ville de départ et la ville d'arrivée. Ce facteur exponentiel risque d'être cependant souvent présent dans des graphes fortement connexes.

3.2 Conclusion

methode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficace

4 Analyse experimentale

4.1 Introduction

Tous nos tests sont effectués en respectant les démarches suivantes :

- Les relevés temporels sont ceux de l'utilisation CPU. Dans un soucis de minimiser les facteurs faussant les relevés (impacts d'autres processus actifs), nous avons essayé d'avoir un minimun de processus actifs pendant les relevés.
- Chaques tests a été effectué 3 fois. Le resultat final étant la moyenne de ces trois tests.
- Les tests ont pour objectif de confronter nos raisonement lors de l'analyse théorique (cf :)

Le premier test consiste à étudier les temps d'exécution des deux méthodes selon le nombre de nœuds et la structure choisie. La densité est fixé à 0,5 soit $0.5 * N^3$ arcs. Le paramètre A est fixé à 1 pour éviter les chemins absorbants et K est aussi fixé à 1 Le second test consiste à étudier les temps d'exécution de la méthode détour borné en faisant varier K. La structure est une liste, la densité fixé à 0,5. Les tests ont été réalisés sur différents nombre de nœuds.

4.2 Étude comparative

4.2.1 Premier test

Pour le premier test (Annexe 1), le fait de fixer A à 1 n'est pas contraignant pour la complexité puisque le temps de calcul sera le même (ou du moins négligeable) pour toute valeur de A. Fixer la valeur de K est bien plus contraignant comme le montrera le second test mais cela permet de voir que, pour une borne minimale, le facteur exponentiel disparaît presque totalement (ce pourrait ce pendant ne pas être le cas, en particulier si tous les chemins ont une valeur proche du PCC. Pour le premier test nous devrions donc avoir un coût proche de celui de Bellman-Ford c'est à dire en O(A*N) soit d'après notre densité $O(N^3)$ or l'on peut voir que lorsque nous multiplions par deux le nombre de nœuds, nous multiplions par dix le temps d'exécution. On peut supposer que cette évaluation est juste. On peut remarquer que les deux méthodes sont très proches et que leur coût provient

essentiellement de Bellman-Ford. Enfin on peut observer une légère différence entre la structure en Liste et en Matrice, on peut supposer que cela provient du coût lors de l'appel de listeArcs.

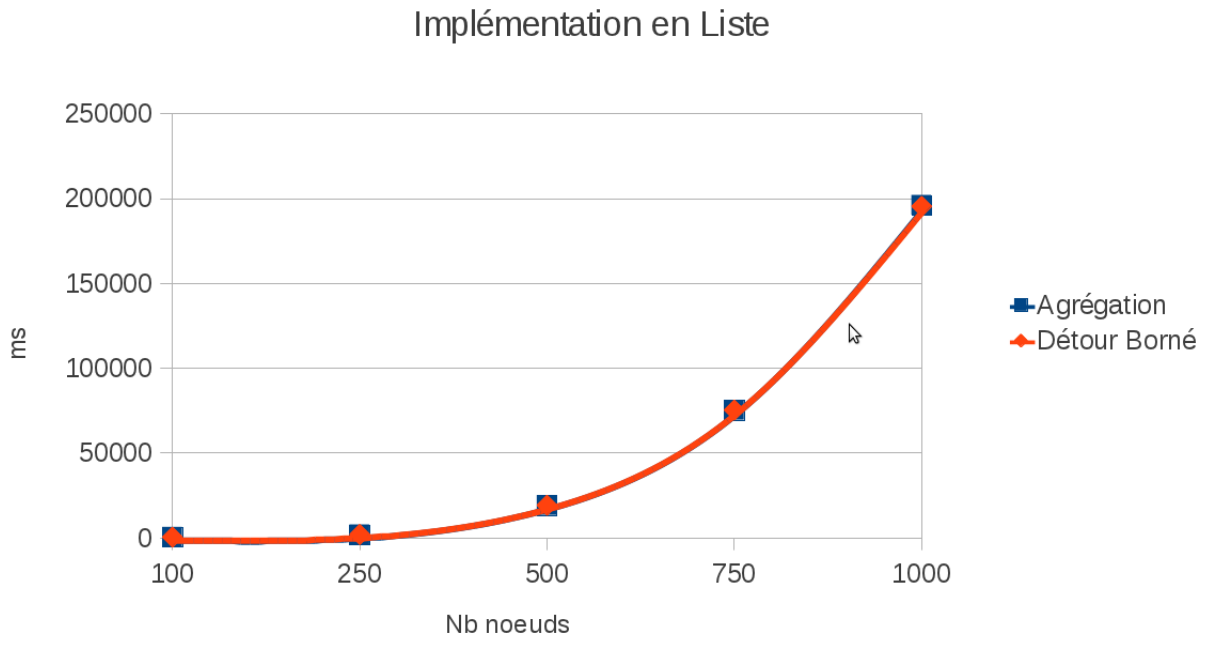
4.2.2 Second test

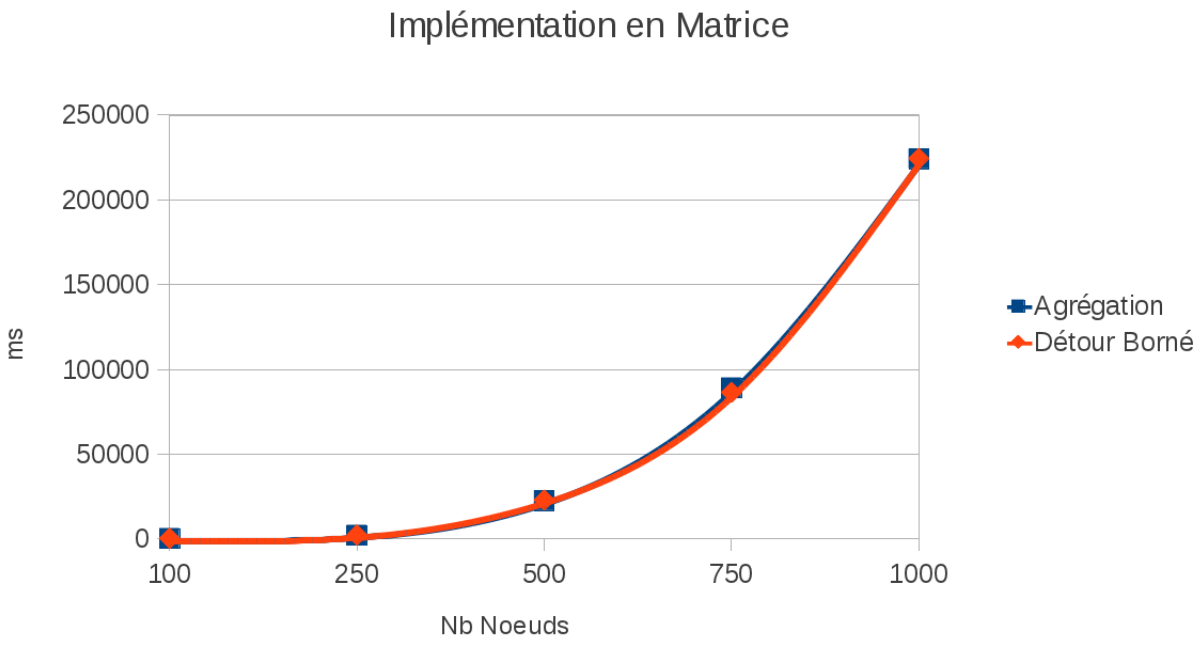
Le second test (Annexe 2) fait clairement ressortir le facteur exponentiel de la méthode détour borné. Lorsque K augmente, l'algorithme va mettre plus de temps pour s'arrêter étant donné qu'il y aura nécessairement des chemins plus long qui ont été ignoré lors d'un parcours avec un K faible. Les tests ont été effectués en faisant varié le nombre de nœuds, bien que dispensable il mettent en évidence le facteur sus-nommé. Cependant le premier test effectué met en évidence que l'effet de K sur la complexité n'est pas continue et qu'à partir d'une valeur donné l'algorithme mettra toujours le même temps pour s'exécuter. K joue donc un rôle d'amortisseur sur la complexité, celle-ci ne dépendant que du nombre de chemins.

5 Annexe

5.1 Annexe 1

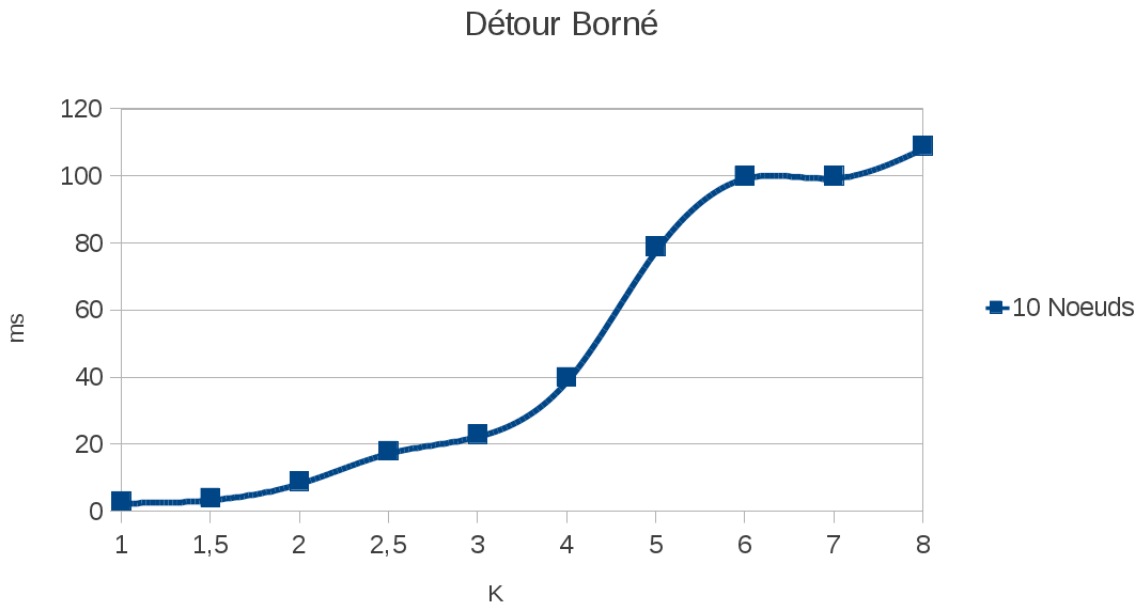
	Liste					Matrice				
	100	250	500	750	1000	100	250	500	750	1000
Agrégation	358	1747	19030	75174	196088	350	2264	22567	89187	224096
Détour Bomé	366	1931	19123	75270	195395	366	2313	22971	86441	224358

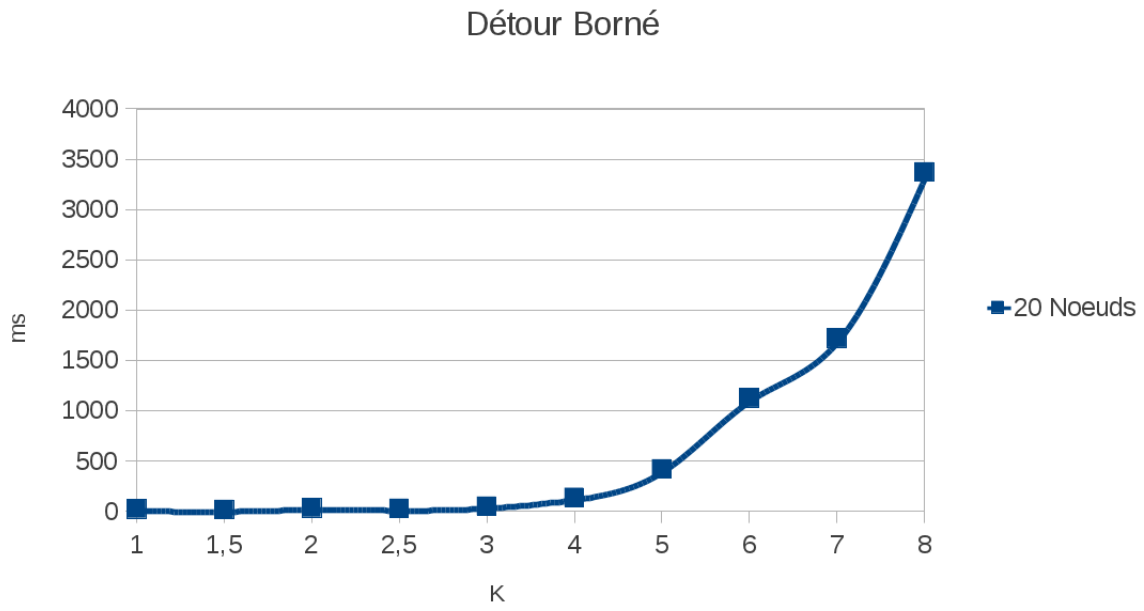


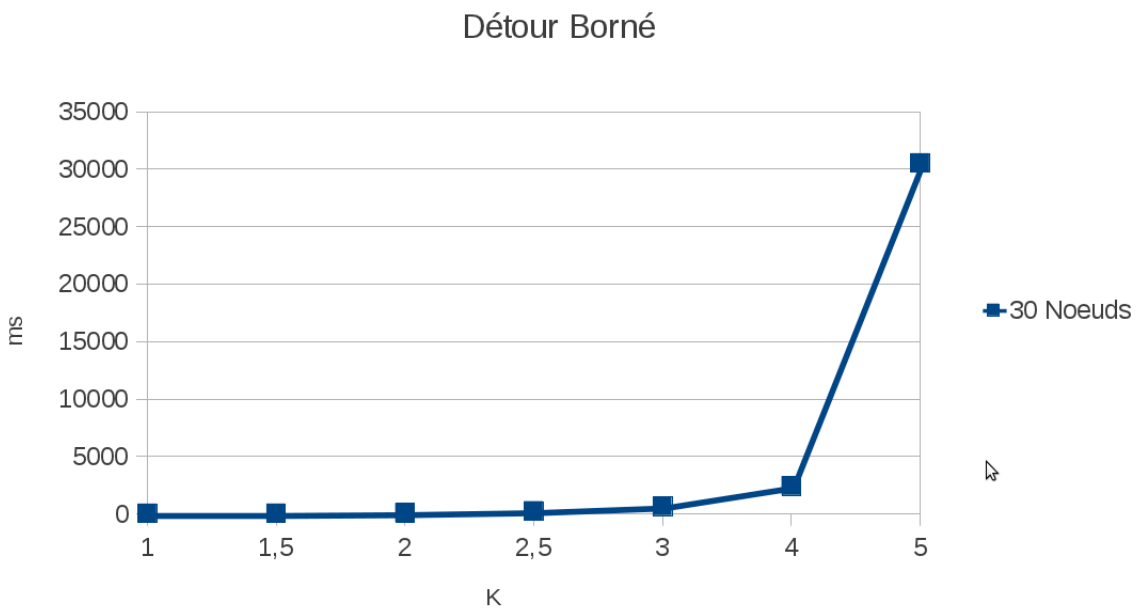


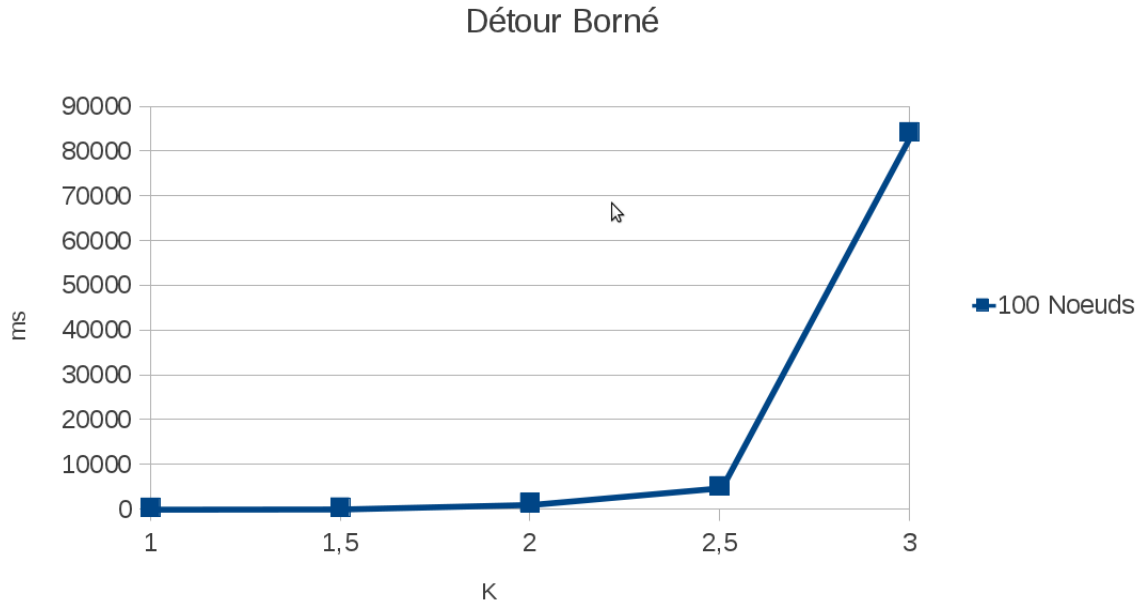
5.2 Annexe 2

K	Détour Borné			
	10 Noeuds	20 Noeuds	30 Noeuds	100 Noeuds
1	3	24	49	366
1,5	4	16	58	403
2	9	34	87	1420
2,5	18	27	264	5158
3	23	49	674	84234
4	40	136	2436	
5	79	420	30525	
6	100	1126		
7	100	1721		
8	109	3372		


Détour Borné
120
100
80
60
40
20
0
ms
1
1,5
2
2,5
3
4
5
6
7
8
K
10 Noeuds


Détour Borné
4000
3500
3000
2500
2000
1500
1000
500
0
ms
1
1,5
2
2,5
3
4
5
6
7
8
K
20 Noeuds


Détour Borné
35000
30000
25000
20000
15000
10000
5000
0
ms
1
1,5
2
2,5
3
4
5
K
30 Noeuds


Détour Borné
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
ms
1
1,5
2
2,5
3
K
100 Noeuds