Ismail Badaoui et Jihed Zenkri

Encadré par Jean Baptiste Yunès

L3 MI

Mesure du temps d'exécution d'algorithme

Année universitaire : 2021/2022

Table des matières

1 Introduction

1.1 Sujet

1.1.1 Phase 1

Le projet consiste tout d'abord à implémenter un certain nombre d'algorithmes (au minimum 4). Le choix des algorithmes est laissé à l'appréciation du groupe qui réalisera le projet mais on suggère très fortement de choisir des algorithmes relativement classiques comme les tris ou certains algorithmes de graphes (ou un mélange de deux).

1.1.2 Phase 2

En seconde phase, il est demandé d'être capable de générer des données d'entrées pour ces algorithmes avec une variabilité contrôlable nécessaire (par exemple la connexité du graphe, le nombre d'éléments d'un tableau ou d'une liste à trier, etc). Attention, générer des données n'est pas toujours aussi trivial qu'il y paraît (en particulier pour les graphes).

1.1.3 Phase 3

En troisième phase, il est demandé d'introduire des mesures de temps d'exécution des différents algorithmes sur des jeux de données et de générer des fichiers de trace de ces temps.

1.1.4 Phase 4

Dans cette phase, il s'agit de présenter les résultats obtenus précédemment permettant de faire apparaître sous la forme de graphiques les temps d'exécutions (temps minimal, temps maximal, temps moyen) comparés des algorithmes implantés (le graphe de t(n)) ainsi que les courbes de référence de leurs complexités analytique et connues.

1.2 Mesure de temps des algorithmes

Pour mesurer le temps d'exectution des algorithmes, on a décidé de fixer un nombre d'échantillon *ech*, la taille maximal du tableau(resp. graphe) m et le pas, donc on prend *ech* tableaux(resp. graphe) de taille p^k avec $k \in \mathbb{N}$,$p \in \mathbb{P}$ tel que $k \leq \log_p(m)$ et on calcule la moyenne du temps pris pour chacun des tableaux(resp. graphe).

2 Tri

2.1 Generation de tableau aléatoire

2.1.1 Tableau aléatoire

Définition 2.1. On définit notre procédure randomTab de paramètres entiers n , a et b (avec $a \leq b$), la création d'un tableau t de taille n où l'élément à l'indice i est entre a et

b, i.e. $t_i \in [\![a, b]\!]$.

```
# randomTab prend des entiers n, a et b et renvoie un tableau
# aleatoire de taille n contenant des entiers compris entre
# les bornes a et b.
def randomTab(n, a, b):
    T = [0] * n
    for i in range(len(T)):
        x = random.randint(a, b)
        T[i] = x
    return T
```

2.1.2 Permutation aléatoire

Définition 2.2. On définit notre procédure randomPerm de paramètre entier n, la création d'une permutation t de taille n, et à l'itération i on permute t_i et t_j si $i \neq j$ (avec $j \in [\![i, n-1]\!]$)

```
# randomPerm prend en parametre un entier n et renvoie une
# permutation aleatoire de longueur n
def randomPerm(n):
    T = [0] * n

    for p in range(n):
        T[p] = p + 1

        for i in range(len(T)):
            x = random.randint(i, n - 1)
            if (x != i):
                tmp = T[i]
                T[i] = T[x]
                T[x] = tmp

    return T
```

2.2 Algorithme de tri

2.2.1 Tri bulles

Définition 2.3. Le principe du tri à bulles est de comparer deux à deux les éléments e_1 et e_2 consécutifs d'un tableau et d'effectuer une permutation si $e_1 > e_2$. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

```
def triBulles(L):
    n = len(L)
    compar = 0
    cng = 0
    for i in range(n):
        for j in range(0, n-i-1):
```

```
            if L[j] > L[j+1] :
                L[j], L[j+1] = L[j+1], L[j]
                cng += 1
            compar +=1
```

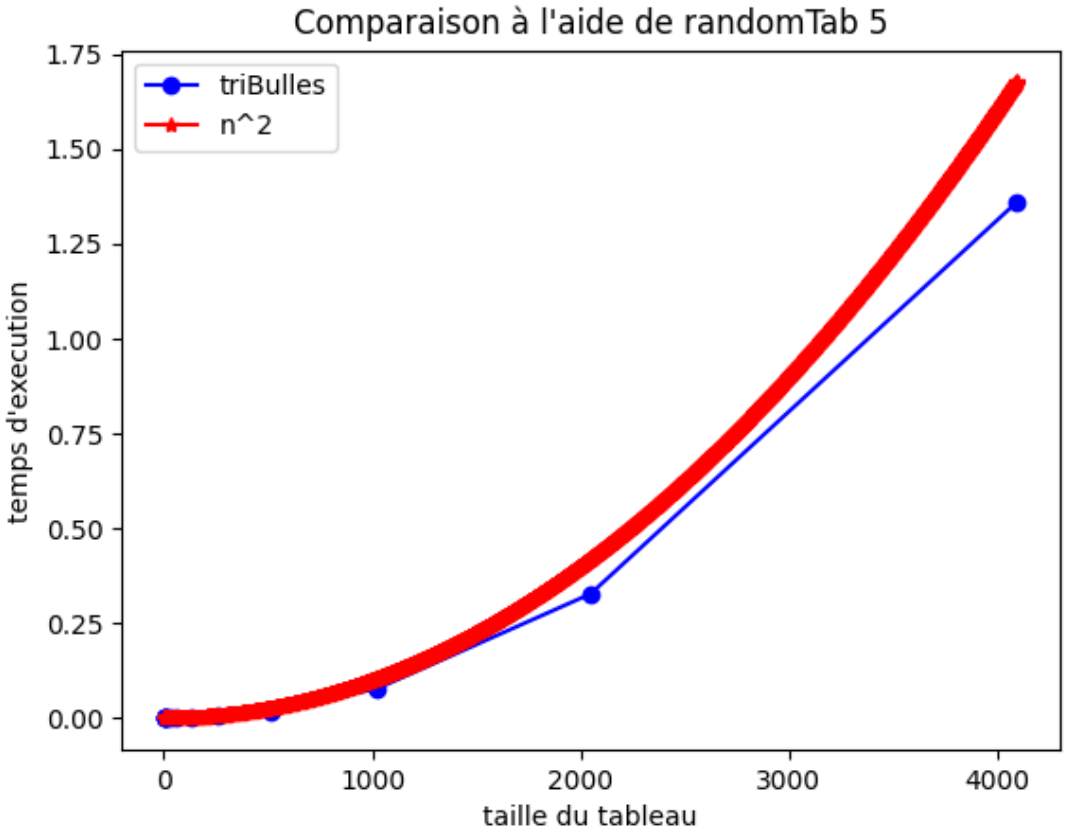


FIGURE 1 – Comparaison de la complexité temporelle du tri bulles et la fonction n^2

2.2.2 Tri insertion

Définition 2.4. Le tri par insertion considère chaque élément du tableau et l'insère à la bonne place parmi les éléments déjà triés. Ainsi, au moment où on considère un élément, les éléments qui le précèdent sont déjà triés, tandis que les éléments qui le suivent ne sont pas encore triés.

```
def triInsertion(L):
    n=len(L)
    compar = 0
    cng = 0
    for i in range(1,n):
        m,k = L[i],i
```

```
        while k>0 and L[k-1]>m:
            L[k],k = L[k-1],k-1
            compar += 2
            cng += 1
        L[k]=m
```

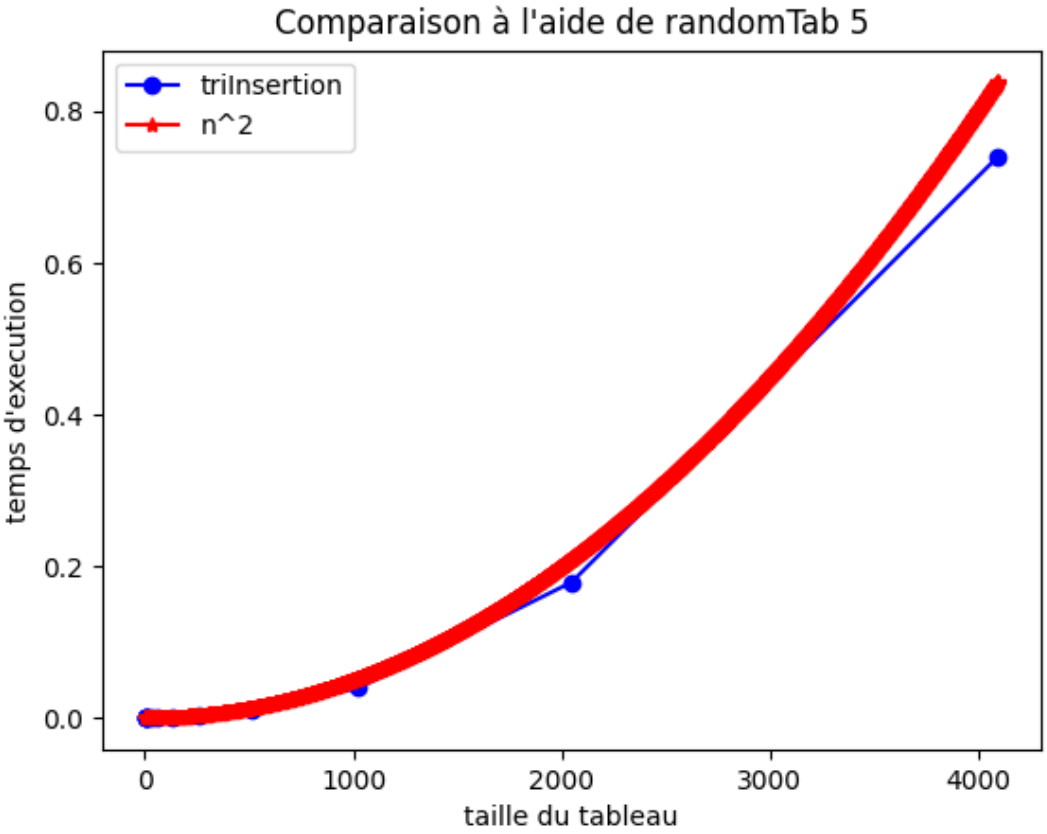


FIGURE 2 – Comparaison de la complexité temporelle du tri insertion et la fonction $\frac{n*(n+1)}{2}$

2.2.3 Tri selection

Définition 2.5. Le tri par sélection consiste à rechercher le minimum de la liste, et le placer en début de liste puis recommencer sur la suite du tableau.

```
def triSelection(T):
    n=len(T)
    compar = 0
    cng = 0
    for i in range(len(T)):
        min = i
        for j in range(i + 1, len(T)):
```

```
8               if T[min] > T[j]:
9                   min = j
10              compar += 1
11          T[i], T[min] = T[min], T[i]
12          cng += 1
```

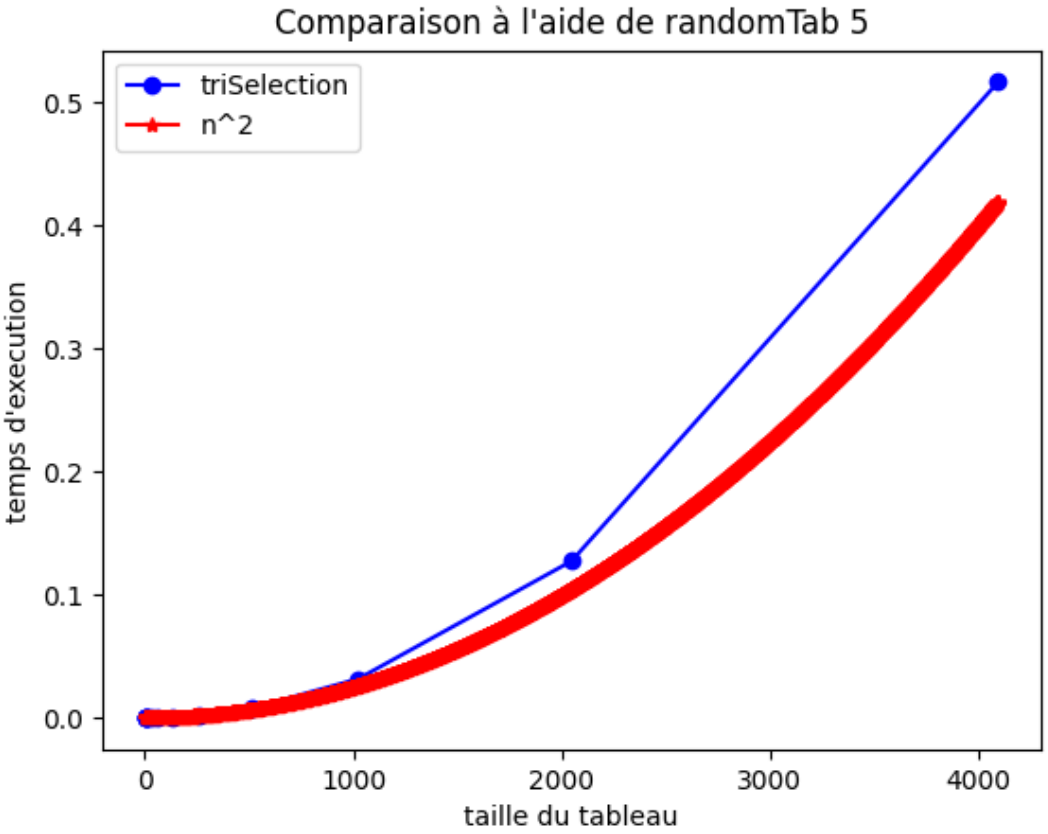


FIGURE 3 – Comparaison de la complexité temporelle du tri selection et la fonction $\frac{n*(n+1)}{4}$

2.2.4 Tri rapide

Définition 2.6. Le tri rapide consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite.

```
def triRapide(T):
    if T == []:
        return []
    else:
        pivot = T[0]
    t1 = []
    t2 = []
    for x in T[1:]:
        if x < pivot:
            t1.append(x)
        else:
            t2.append(x)
    return triRapide(t1) + [pivot] + triRapide(t2)
```

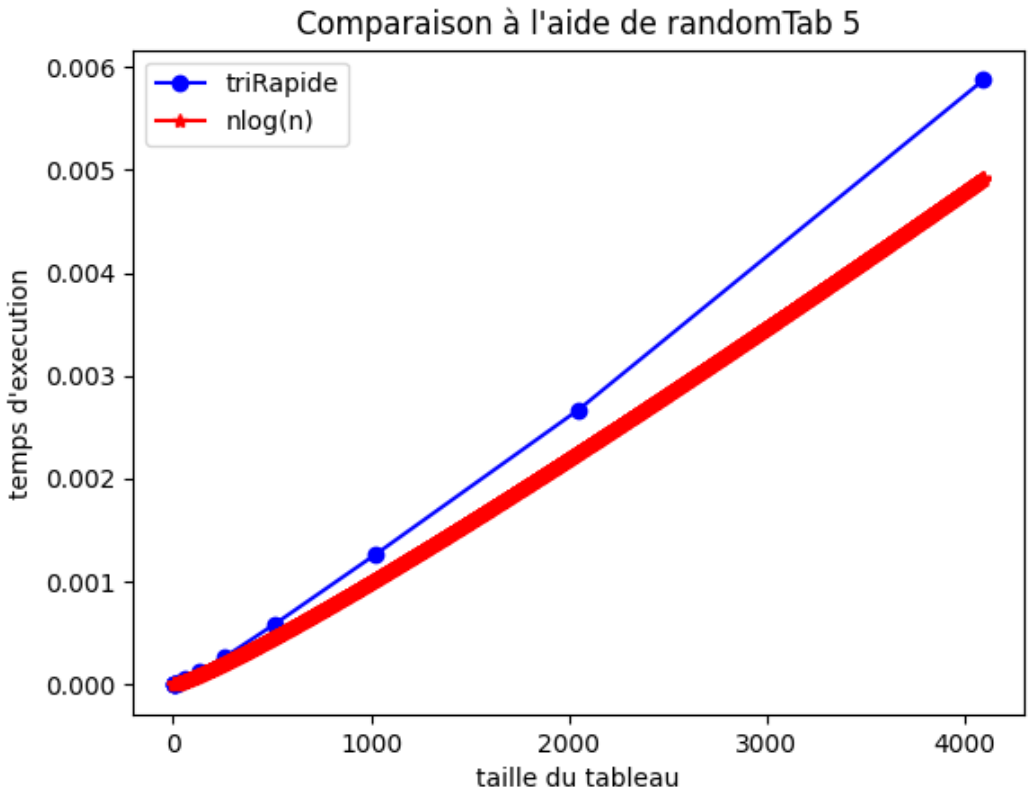
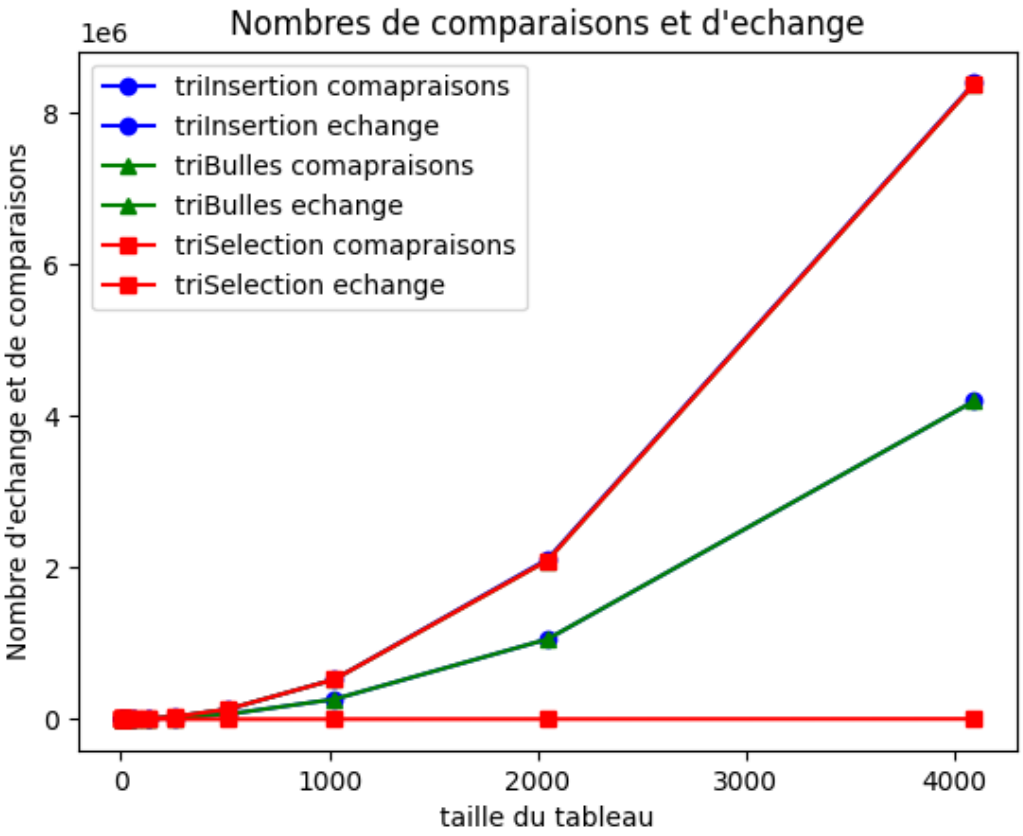
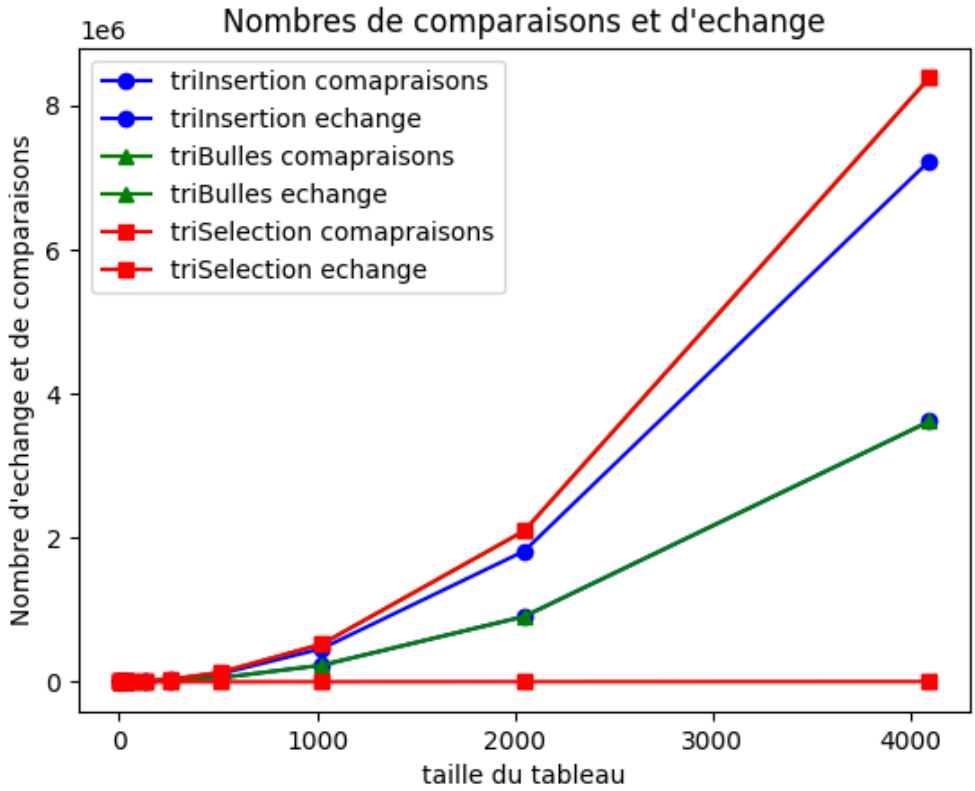


FIGURE 4 – Comparaison de la complexité temporelle du tri rapide et la fonction *nlogn*

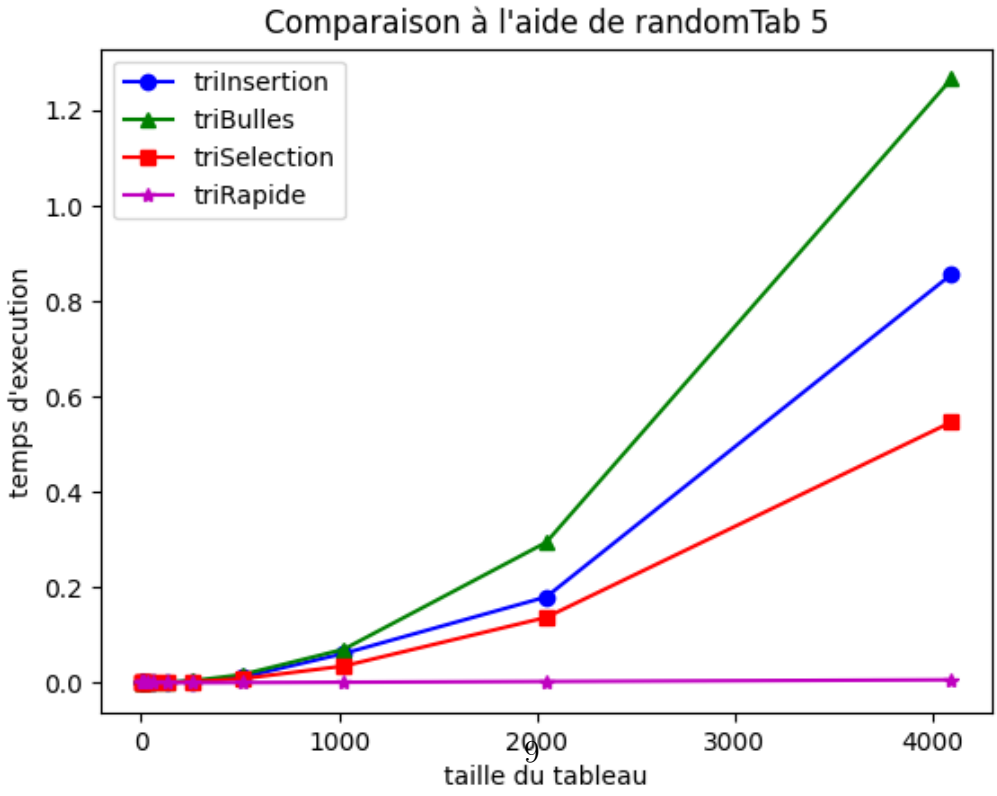
2.3 Mesures et comparaisons



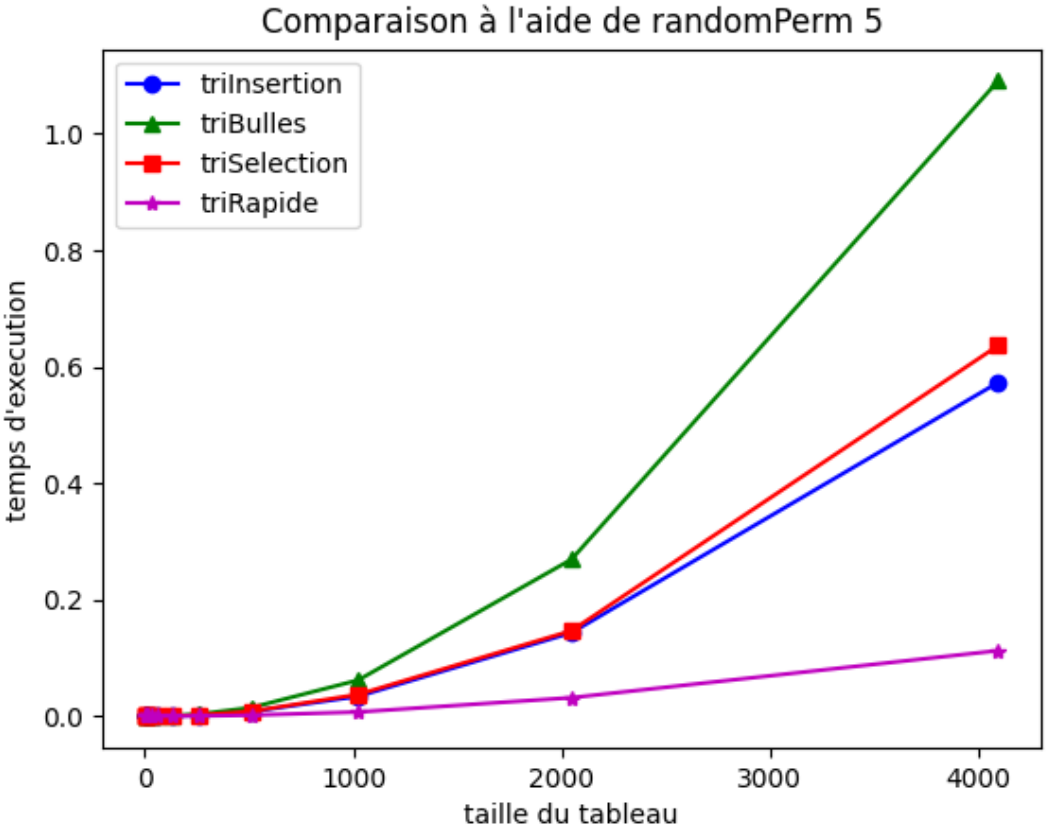
(a) Mesures de nombre de comparaisons et d'échanges des différents algorithmes de tri appliqués sur un tableau aléatoire



(b) Mesures de nombre de comparaisons et d'échanges des différents algorithmes de tri appliqués sur une permutation aléatoire



(c) Mesures de complexité temporelle des différents algorithmes de tri appliqués sur un tableau aléatoire



(d) Mesures de complexité temporelle des différents algorithmes de tri appliqués sur une permutation aléatoire

3 Graphes

3.1 Generation de graphes aléatoire

3.1.1 Triangulation de Delaunay

Définition 3.1 (Triangulation de Delaunay)**.** La triangulation de Delaunay d'un ensemble de $n \geq 3$ points est l'unique traingulation telle qu"un cercle passant par les trois points d'un triangle ne contienne aucun autre point.

Algorithme de flip

1. Insèrer tous les côté de la triangulation initiale T dans une pile et le marquer comme appartenant à la pile.
2. Tant que la pile n'est pas vide
   - Dépiler le premier côté st et le démarquer
   - Si st est illégal
     - Soient rst et pst les deux triangle de part et d'autre de st
     - Remplacer st par rp dans T
     - Marquer et empiler ceux de côtés rs, rt,ps et pt qui ne sont pas dèjà marqués[1]

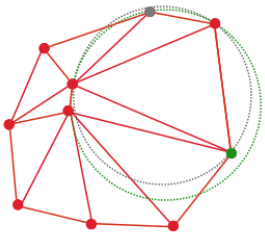


FIGURE 6 – Triangulation quelconque

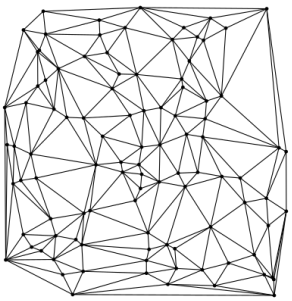


FIGURE 7 – Triangulation de Delaunay

3.1.2 Fast generation of random connected graphs with prescribed degrees

Définition 3.2 (Test d'isolation)**.** Un test d'isolation de largeur K sur un sommet v teste si ce sommet appartient à une composante connexe de taille inférieure ou égale à K.[2]

Définition 3.3 (Largeur d'isolation caractéristique)**.** La largeur d'isolation caractéristique K_G d'un graphe G contenant m arêtes est la largeur minimal K du test d'isolation tel que la probabilité que le graphe soit non connexe $p(K)$ vérifie $p(K) < 1/m$[2]

Lemme 3.1. L'application de la procédure du shuffle sur un graphe G contenant au moins 10 arêtes, avec un test d'isolation de largeur $K \geq K_G$, et une periode $T = m$ à un taux de réussite $r > 1/3$[2]

Lemme 3.2. Pour un degré de distrbution donné, la largeur d'isolation caractéristique K_G d'un graphe aléatoire de taille n est dans $O(logn)$[2]

Theorème 3.3. Pour un degré de distribution donné, le proccesus du shuffle pour un graphe de taille n a une complexité temporelle de O($nlogn$) et spactiale de O(m) [2]

Preuve On définit notre procédure shuffle(G) comme suit :

1. On pose $K \coloneqq 1$
2. On sauvegarde le graphe G
3. On échange m arêtes du graphe G avec des tests d'isolation de largeur K
4. Si le graphe est connexe , on le return
5. Sinon on restaure le graphe G à sa valuer sauvgardée , on pose $K \coloneqq 2 * K$ et on recommence à partir de l'étape de 2.

Cette procèdure nous renvoie un graphe connexe obtenue après l'échange de m arête de G. Les lemme 1 et 2 assurent que cette procédure prend fin après $O(loglogn)$ iterations. De plus le coût de l'iteration i est $O(2^i * m)$, d'où la complexité globale est $O(mlogn)$, or $m = O(n)$, donc la complxité temporelle est $O(nlogn)$

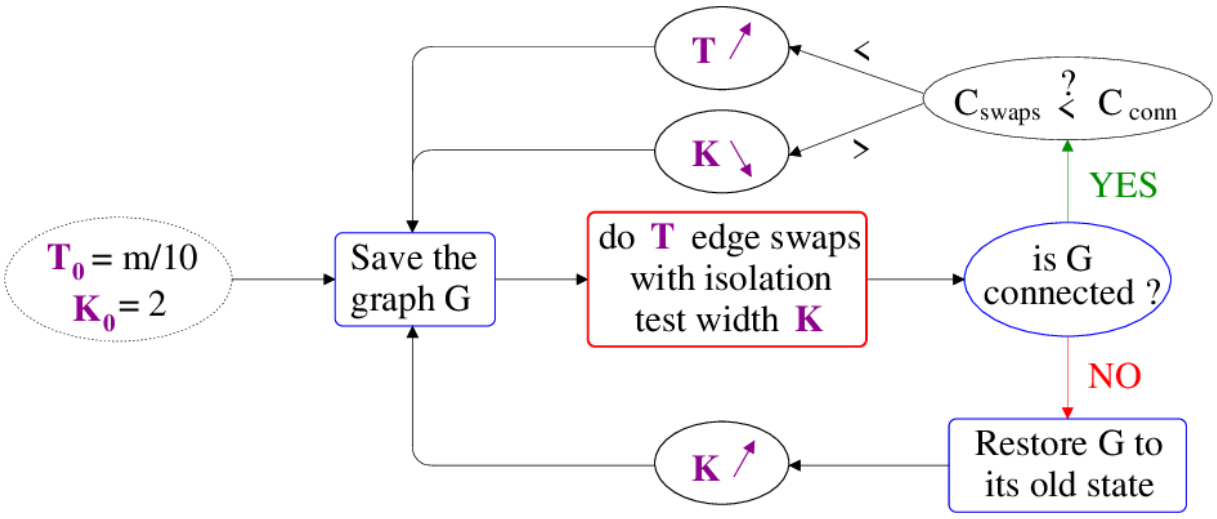


FIGURE 8 – Schèma représentant la procédure shuffle

3.2 Algorithme de graphe

3.2.1 Dijkstra

Définition 3.4. Soit $G = (E, V)$ un graphe orienté avec pondération $l \in \mathbb{R}^+$ et $s \in S \subseteq V$. Pour tout $u \in V$, on note $d(u) = dist(s, u)$. Pour tout sommet $v \notin S$, on définit $D(v) = min\{d(u) + l(u, v) : u \in Set(u, v) \in E\}$

Lemme 3.4 (Principe de sous-optimalité)**.** Si P est un court chemin de s vers v alors, en notant v' le prédécesseur de v dans ce chemin, le sous chemin de P qui va de s vers v' est un plus court chemin de s vers v'.

Lemme 3.5. Soit v_0 tel que $D(v_0)$ est minimum, parmi tous les sommets dans V S. Alors, $D(v_0) = d(v_0)$.

Preuve

— Soit v_0 tel que $D(v_0)$ est minimum.
— Comme $D(v_0)$ et la longuer d'un chemin de s à v_0, on a $D(v_0) \geq d(v_0)$.
— Si $D(v_0) \leq d(v_0)$ alors la preuve est terminé
— Sinon, on suppose que $D(v_0) > d(v_0)$.
— Soit P un plus court chemin de s vers v_0 (son poids est alors $d(v_0)$).
— Soit x le premier sommet de P qui n'appartient pas à S.
— Soit $y \in$ le prédécesseur de x dans ce chemin.
— Soit P_y le sous chemin de P de s vers y.
— P_y est un plus court chemin de s vers y (principe de sous optimalité).
— Soit P_x le sous chemin de P de s vers x.
— La longuer de P_x est $d(y) + l(y, x)$
— Ceci entraîne que $D(x) \leq d(y) + l(y, x) \leq d(v_0) < D(v_0)$.
— Implique $D(x) < D(v_0)$ contradiction avec le choix de $D(v_0)$

```
def dijkstra(graphe,start):
    distance = [float("infinity")]*graphe.vertices
    distance[start] = 0
    priorityqueue = [(distance[start],start)]
    while(len(priorityqueue)>0):
        curr_cost,curr_node = heapq.heappop(priorityqueue)
        if(curr_cost>distance[curr_node]):
            continue
        for neighbor,w in graphe.edges[curr_node]:
            new_cost = curr_cost+w
            if(new_cost<distance[neighbor]):
                distance[neighbor] = new_cost
                heapq.heappush(priorityqueue,(new_cost,neighbor))
    return distance
```

Complexité de l'algorithme de Dijkstra : L'algorithme de Dijkstra est similaire au BFS, cependant, il est plus lent car les files de priorité sont plus exigeantes, et puisque le makequeue prend au plus autant de temp que $|V|$ opérations d'insertion, nous obtenons un total de $|V|$ heappop et $|V| + |E|$ opérations d'insertion/heappush, d'où notre complexité est de $O(m + nlogn)$

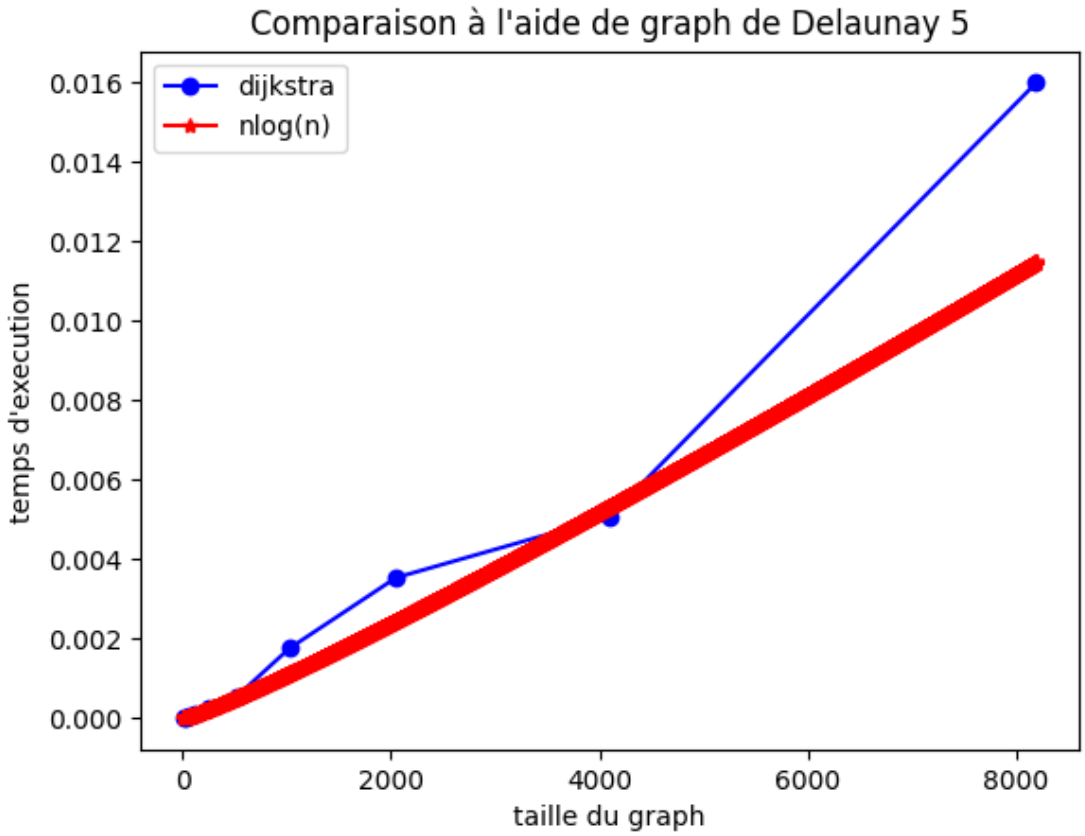


FIGURE 9 – Comparaison de la complexité temporelle de Dijkstra et la fonction $n+nlogn$

3.2.2 Bellman-Ford

Lemme 3.6. Après i itérations de la boucle principale :

- Si $D[u] \neq +\infty$ alors $D[u]$ est la longueur d'un chemin de s à u.
- S'il existe un chemin de s à u comprenant au plus i arcs, alors la valeur de $D[u]$ est inférieure ou égale à la longueur d'un plus ourt chemin de s à u comprenant au plus i arcs.

```
def bellman_ford(graphe,start):
    distance = [float("infinity")]*graphe.vertices
    distance[start] = 0
    for _ in range(graphe.vertices-1):
        for u in range(graphe.vertices):
            if(u==start):
                continue
            for v,w in graphe.edges[u]:
                if(distance[u]!=float("infinity") and distance[u]+w<
   distance[v]):
                    distance[v] = distance[u]+w
    return distance
```

Complexité de l'algorithme de Bellman-Ford : $O(m * n)$
Détéction de cycles négatifs : Un cycle négatif existe dans G si et seulement si il y au moins un changement dans le tableau D lors de l'itération qu'on ajoute après avoir fait les $|V| - 1$ itérations de la boucle.

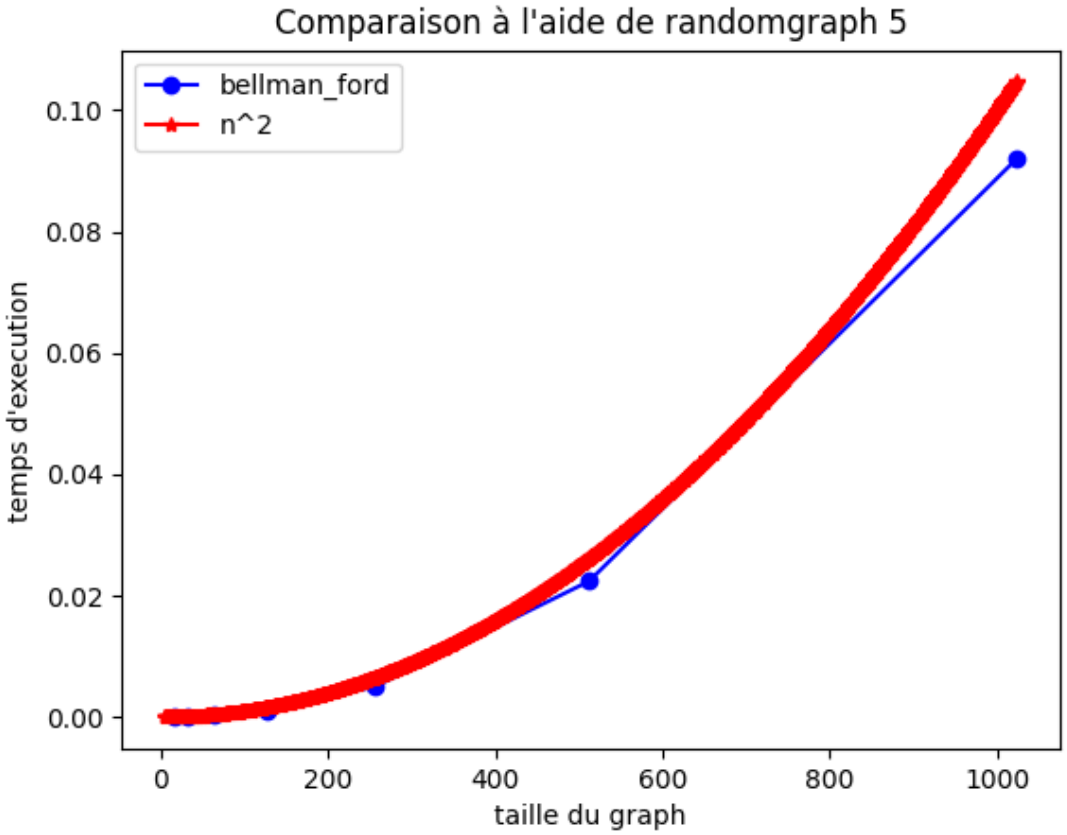


FIGURE 10 – Comparaison de la complexité temporelle de Bellman-Ford et la fonction n^2

3.2.3 Floyd-Warshall

```
def floyd_warshall(graphe , start):
    distance = [[float("infinity") for i in range(graphe.vertices)] for
   j in range(graphe.vertices)]
    for i in range(graphe.vertices):
        for j,w in graphe.edges[i]:
            distance[i][j] = w
    for k in range(graphe.vertices):
        for i in range(graphe.vertices):
            for j in range(graphe.vertices):
                distance[i][j] = min(distance[i][j],distance[i][k]+
   distance[k][j])
```

```
10      return distance
```

Complexité de l'algorithme de Floyd-Warshall : $O(n^3)$

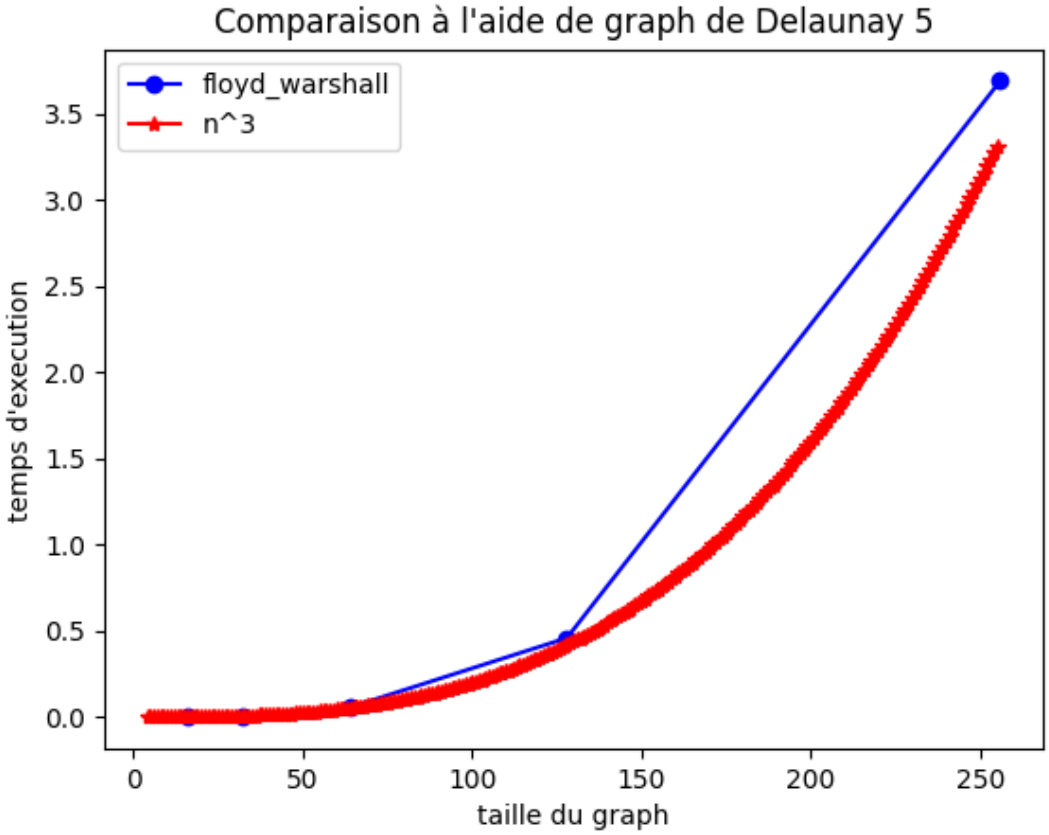


FIGURE 11 – Comparaison de la complexité temporelle de Floyd-Warshall et la fonction n^3

3.2.4 Kruskal

Définition 3.5. Soit $G = (E, V)$ un graphe connexe avec une pondération $w \in \mathbb{R}^{|E|}$. Un arbre couvrant de poids minimum de G est un arbre couvrant $T \subseteq G$ qui minimise $w(T) = \sum_{e \in E} w_e$

Lemme 3.7. Soit x un sommet d'une arborescence A. Si $rank(x) = k$, alors la sous arborescence avec racine x a au moins 2^k sommets.

Preuve

— Vrai pour $k = 0$

— Supposons que la propositioon est vraie pour un entier $k \geq 0$, et soit x un sommet d'une arborescence A telle que $rank(x) = k + 1$

— On a $rank(x) = k + 1$ parce que l'on fusionné deux arborescences A_1,A_2, la racine de chacune ayant rang k
— Par l'hypothèse de récurrence, A_1 et A_2 ont donc chacune au moins 2^k noeuds
— A a donc au moins $2 * 2^k = 2^{k+1}$ sommets
— D'où $rank(x) \leq log_2 n$ pour tout sommet x
— Donc find et union sont de complexité $O(logn)$

```
def find(graphe, parent, i):
        if parent[i] == i:
            return i
        return find(graphe,parent, parent[i])

def apply_union(graphe, parent, rank, x, y):
        xroot = find(graphe,parent, x)
        yroot = find(graphe,parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
def kruskal_algo(graphe,start):
        result = []
        i, e = 0, 0
        graphe = sorted(graphe, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(graphe.vertices):
            parent.append(node)
            rank.append(0)
        while e < graphe.vertices - 1:
            u, v, w = graphe.edges[e][0]
            i = i + 1
            x = find(graphe,parent, u)
            y = find(graphe,parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                apply_union(graphe,parent, rank, x, y)
```

Correction de l'algorithme de Kruskal : L'algorithme d'arrête au pire lorsque toutes les arêtes dans E ont été considérées, Comme E est fini, l'algorithme s'arrête au bout d'un nombre fini d'opérations élémentaires. Soit $X \subseteq E$ la sortie de l'algorithme donc $T = (V, X)$ un sous graphe couvrant de G, puisqu'il contient tous les sommets de G. Supposons par l'absurde que T n'est pas connexe, soient T_1 et T_2 deux composantes connexes de T et soit e une arête dans X entre T_1 et T_2 de poids minimum (car G est connexe), comme $(V, X \cup \{e\})$ est acyclique, l'algorithme aurait ajouté e à X, donc T est connexe, de plus T est acyclique par la condition de la boucle, d'où T est un arbre couvrant de G, soit $T_0 = (V, X_0)$ un arbre couvrant de G de poids minimum, tel que $|X \cap X_0|$ soit maximum. Supposons par l'absurde que $|X \cap X_0| < |X|$, et soit e_1 l'arête

la plus légère dans $X_0 \setminus X$, alors le graphe $(V, X \cup \{e_1\})$ contient un cycle C, comme T_0 est acyclique, il existe au moins une arête $e_2 \in E(C) \setminus X_0$, si $w_{e1} < w_{e2}$, l'algorithme de Kruskal aurait choisi l'arête e_1 au lieu de e_2 donc $w_{e1} \geq w_{e2}$. Soit $X_1 = (X_0 \setminus \{e_1\}) \cup \{e_2\}$, comme $w(X_1) \leq w(X_0)$, (V, X_1) est un autre arbre couvrant de G de poids minimum, tel que $|X \cap X_1| > |X \cap X_0|$ contradiction avec l'hypothèse donc $X = X_0$ et on conclut que $T = (V, X)$ est un arbre couvrant de G de poids minimum.

Complexité de l'algorithme de Kruskal :$O(mlogn)$

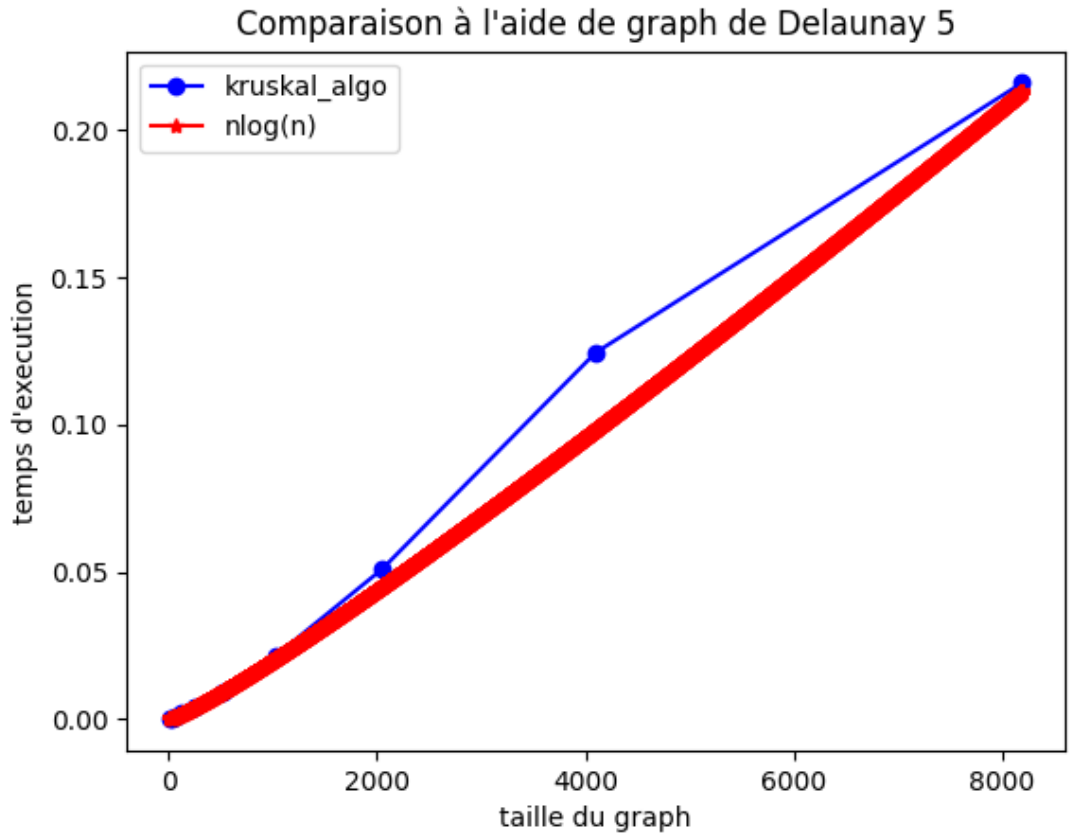


FIGURE 12 – Comparaison de la complexité temporelle de Kruskal et la fonction $20nlogn$

3.2.5 A^*

Définition 3.6. Une heuristique h est **admissible** si pour tout noeud u, $h(u)$ est une borne inférieure de la plus courte distance séparant u de la cible t, i.e. $h(u) \leq D(u,t)$ (avec $D(u,t)$ désignant la longueur d'un chemin optimal entre u et t).[3]

Définition 3.7. Une heuristique h est **consistante** si pour toute arête $e = (u, v)$ nous avons $h(u) \leq h(v) + w(u, v)$.[3]

Définition 3.8. Soient $(u_0, \ldots, u_k)$ un chemin et $g(u_i)$ le coût du chemin $(u_0, \ldots, u_i)$. Nous posons $f(u_i) = g(u_i) + h(u_i)$. Une heuristique h est **monotone** si pour tout $0 \leq i < j \leq k$ nous avons $f(u_j) \geq f(u_i)$. C'est à dire que l'estimation du poids total d'un chemin ne décroît pas lors du passage d'un noeud à ses successeurs[3]

L'algorithme A^* **:**L'algorithme A^* est un exemple d'amélioration de l'algorithme de Dijkstra grâce à l'utilisation d'une fonction heuristique. A^* associe à un noeud u du graphe la valeur :

$$f(u) = g(u) + h(u)$$

où $g(u)$ est le poids optimal actuellement connu pour un chemin menant de s à u, et $h(u)$ est une heuristique admissible
Nous remarquons que pour un noeud u, nous avons pour tout successeur v de u

$$\begin{aligned} f(v) &= g(v) + h(v) \\ &= g(u) + w(u,v) + h(v) \\ &= g(u) + h(u) + w(u,v) - h(u) + h(v) \\ &= f(u) + w(u,v) - h(u) + h(v) \end{aligned}$$

D'où, l'algorithme A^* correspond exactement à l'algorithme de Dijkstra aec la repondération suivante :

$$w(u,v) = w(u,v) - h(u) + h(v)$$

Si l'heuristique h est monotone alors tous les poids restent positifs et donc la preuve de correction de Dijkstra s'applique.[3]

```
def a_star_algorithm(self, start_node, stop_node):
        # open_list est la liste des  nodes visites , mais ses voisins
   ne sont pas inspectes
        # closed_list est la liste des  nodes visites
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])
        # g contient les distances de start_node aux autres
        # la valeur par defaut est +inf
        g = {}
        g[start_node] = 0
        # parents  contient la liste dadjacence des nodes
        parents = {}
        parents[start_node] = start_node
        while len(open_list) > 0:
            n = None
            #  chercher le noeud avec un valeur minimal +evaluation
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v
            if n == None:
                print('Path does not exist!')
                return None
```

```
            # if the current node is the stop_node Si le noeud courant est  le stop_node , on reconstruit le  chemin de ce noeud vers strat_node
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path
            # Pour tous les voisins du noeud courant
            for (m, weight) in self.get_neighbors(n):
                # S'il est pas dans les 2 listes (open et closed) , on l'ajoute a open_list and on note n son parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #Sinon , on verifie si il est plus rapide de visiter n apres m et on maj le parent data et g data ,
                # Si le noeud est dans closed_list , on le place dans open_ist
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)
            #supprimer n d'open_list , et on l'ajoute a closed_list
            open_list.remove(n)
            closed_list.add(n)
        print('Path does not exist!')
        return None
```

Complexité de l'algorithme de A^* :$O(|E|)$

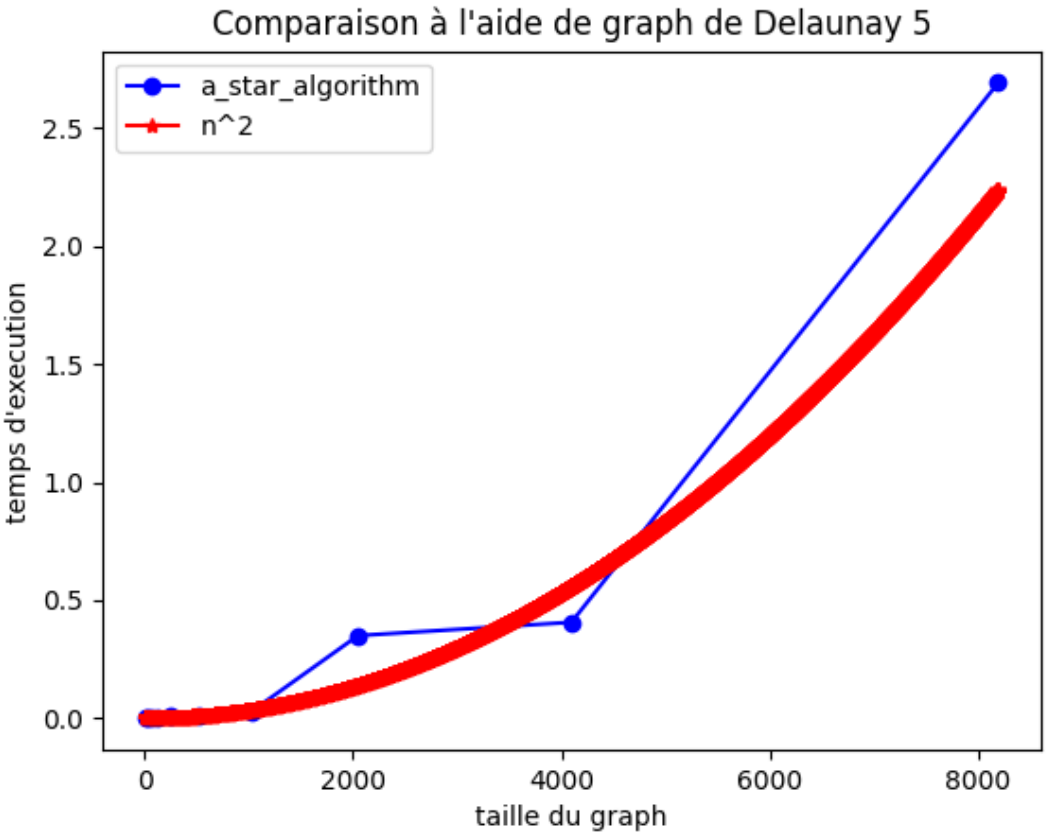
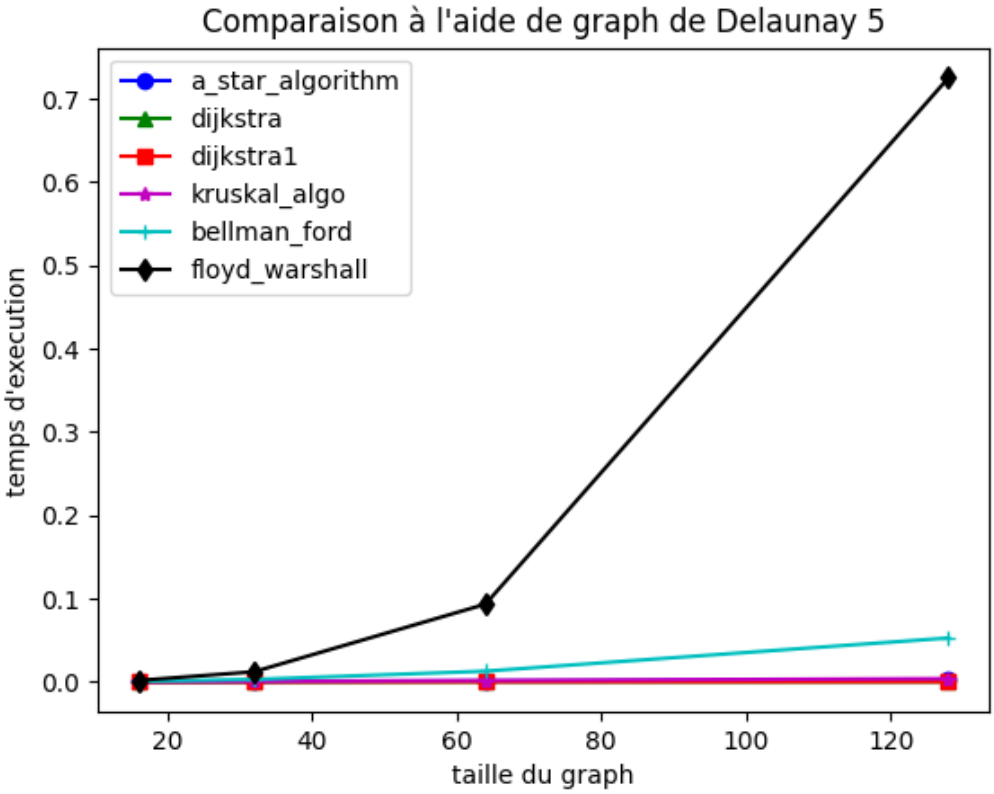
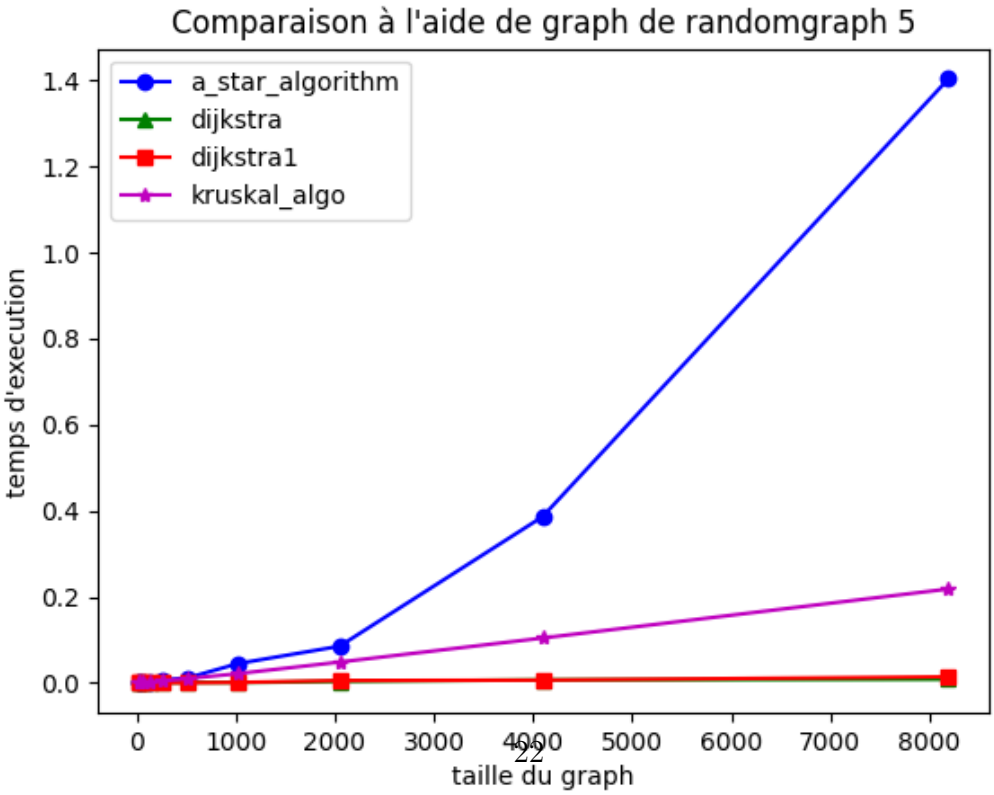


FIGURE 13 – Comparaison de la complexité temporelle de A* et la fonction $\frac{n^2}{3}$

3.3 Mesures et comparaisons



(a) Mesures de complexité temporelle des différents algorithmes de graphes appliqués sur un graphe de Delaunay



(b) Mesures de complexité temporelle des différents algorithmes de graphes appliqués sur un graphe aléatoire

Références

[1] Mathieu Brévilliers. *Construction of the segment Delaunay triangulation by a flip algorithm*. Theses, Université de Haute Alsace - Mulhouse, December 2008.

[2] Fabien Viger and Matthieu Latapy. Fast generation of random connected graphs with prescribed degrees. *arXiv preprint cs/0502085*, 2005.

[3] Pierre-Edouard Portier. https ://perso.liris.cnrs.fr/pierre-edouard.portier/teaching_2014_2015/ia/astar/astar.html.