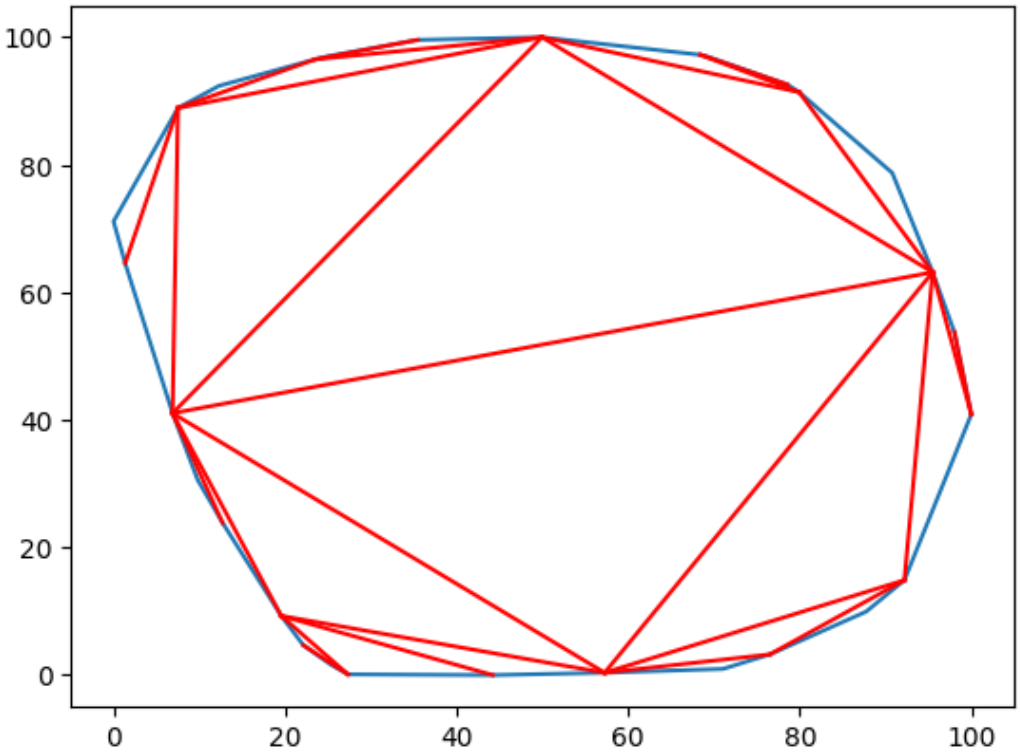
Compte rendu projet d'algorithmique avancée

Université de Rennes

ENSSAT LANNION



Encadrant : MMe Wafa EL HUSSEINI

Enseignant : Mr. Olivier Pivert

étudiants : Yasser JARMOUNI et Haydar SELMI

16/04/2023

Algorithmique avancée S4

INTRODUCTION

Dans ce projet, nous abordons la triangulation d'un polygone convexe, un problème clé dans divers domaines tels que l'infographie, la géométrie algorithmique et la modélisation 3D. La triangulation est essentielle pour le rendu d'objets 3D, car elle permet de convertir les polygones complexes en ensembles de triangles, qui sont plus faciles à traiter et à afficher.

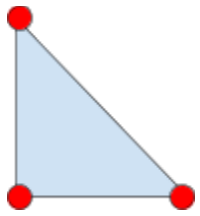
Nous étudions trois méthodes pour résoudre ce problème : les essais successifs, la programmation dynamique et l'algorithme glouton. Chaque méthode présente des avantages et des inconvénients en termes de complexité, d'optimalité et d'efficacité. Nous analysons et comparons ces méthodes pour déterminer la plus adaptée à la résolution du problème dans le contexte de la modélisation 3D et d'autres applications. En conclusion, nous formulerons une recommandation argumentée quant à la méthode la plus appropriée pour résoudre le problème de la triangulation d'un polygone convexe, en tenant compte des performances respectives des différentes méthodes étudiées et de leur pertinence pour les applications en 3D.

A. Questions préliminaires:

1. Nombre de cordes distinctes dans un polygone convexe :

Pour ce faire on va procéder par démonstration par récurrence simple avec des schémas illustrant nos propos.

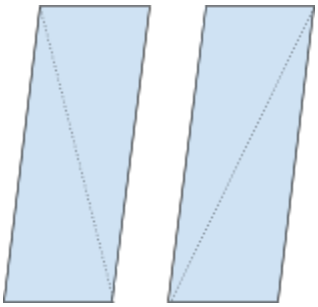
Tout d'abord on a le cas particulier du triangle qui est tout de même un polygone à 3 sommets, dont tous les sommets sont adjacents les uns aux autres donc on remarque que :



- nombres de sommets n = 3 -> il n'y a pas de cordes distinctes.

Le cas du triangle est important pour comprendre la formule de récurrence ensuite.

Ensuite on a le cas du polygone à 4 sommets n = 4 et on remarque qu'on pourra lier que 2 sommets par corde car le polygone est convexe et on ajoutant plus de cordes on est sûr d'avoir un intersection avec celle-ci.



Démonstration par récurrence :

Initialisation :

Pour n = 4 on a une seule avec cette formule n - (le sommet lui-même + 2 autres sommets adjacents) = n - 3 cordes distinctes possibles = 1. On a La propriété vraie pour

Hérédité :

Supposons que la propriété soit vraie pour un polygone convexe à n sommets, c'est-à-dire qu'il peut être triangulé en utilisant n - 3 cordes distinctes sans intersection.

Considérons un polygone convexe à n+1 sommets. En traçant une corde entre deux sommets non adjacents, nous divisons le polygone en deux sous-polygones convexes. D'après notre hypothèse de récurrence, chacun de ces sous-polygones peut être triangulé en utilisant respectivement m - 3 et k - 3 cordes distinctes sans intersection.

En combinant ces triangulations et en ajoutant la corde tracée, nous avons une triangulation pour le polygone à n+1 sommets avec (n+1) - 3 cordes, ce qui prouve la propriété par récurrence.

2. Toutes les triangulations d’un même polygone convexe comportent le même nombres de cordes distinctes :

On peut procéder par récurrence sur le nombre de sommets du polygone.

Pour n = 3, le polygone est déjà un triangle, donc il n'y a pas de diagonales possibles.

Pour n = 4, le polygone est un quadrilatère. Il est possible de le trianguler de deux manières différentes, soit en reliant les sommets opposés, soit en reliant les sommets adjacents. Dans les deux cas, il y a une seule diagonale.

Pour n > 4, supposons que le résultat soit vrai pour tout polygone à k sommets, avec k < n. Considérons maintenant un polygone à n sommets et choisissons un sommet quelconque . On peut diviser le polygone en deux polygones plus petits, en traçant une diagonale à partir de ce sommet vers un autre sommet non consécutif. Le premier polygone a k sommets et le deuxième a n-k sommets. Par hypothèse de récurrence, chaque polygone a le même nombre de cordes dans toutes ses triangulations. Dans la triangulation d'origine du polygone à n sommets, la diagonale que nous avons dessinée est une corde, et chaque triangulation de ce polygone se compose d'une triangulation du premier polygone, une triangulation du deuxième polygone, et la corde que nous avons dessinée. Par conséquent, chaque triangulation du polygone à n sommets est obtenue en combinant une triangulation du premier polygone, une triangulation du deuxième polygone, et la corde que nous avons dessinée. Il y a le même nombre de triangulations pour chaque polygone plus petit, donc il y a le même nombre de possibilités pour combiner les triangulations du premier et du deuxième polygone, et donc le même nombre total de triangulations du polygone à n sommets.

B. Essais successifs

1. Fonction validecorde :

Suppositions : Lors de la création des cordes i < j et p < q et jamais de couples de cordes à 2 sommets identiques. (i, j) différent de (p, q).

fonction intersecte((i, j), (p, q))

Si i == p et q == j Alors retourne vrai

Si i entre [p, q] XOR j entre [p, q] Alors retourne vrai

retourne faux

fin fonction

fonction validecorde(i, j, cordes_tracées)

pour chaque corde (p, q) dans cordes_tracées faire

si la corde (i, j) intersecte la corde (p, q) alors

retourne faux

fin si

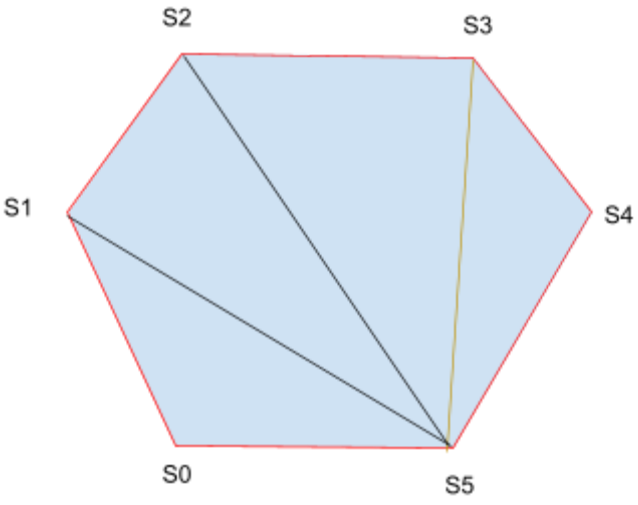
fin pour

retourne vrai

fin fonction

2. Solution possible d'essais successifs non exhaustif :

L'algorithme par essais successifs basé sur cette stratégie peut calculer plusieurs fois la même triangulation si les cordes tracées précédemment sont les mêmes mais dans un **ordre** différent.



D'après nos calculs par essais successifs on remarque que pour n sommets on a toujours n - 3 sommets à tracer, cela va nous permettre de définir qu'on a trouvé une triangulation possible. On peut ainsi en déduire le cas d'arrêt qui est ici

(nombres d'étapes = n - 3). Pour le cas d'arrêt de l'algorithme comme on cherche la triangulation donnant la distance de cordes la plus petite alors on peut admettre le cas d'arrêt où on s'arrête si on trouve pas d'autres triangulation aux prochaines itérations donnant une distance des cordes plus petite, si par exemple au bout d'un certain nombre de triangulations on ne trouve pas de meilleurs triangulations on arrête l'algorithme.

Cette méthode ne permet pas toujours d'obtenir toutes les triangulations, car elle ne prend pas en compte toutes les combinaisons de cordes valides.
Là par exemple S0 n'est pas relié à une corde, or à l'étape 0 il n'y a pas de de corde déjà tracée. Donc il y aura toujours une corde passant par S0 dans toutes les triangulations faites par l 'algorithme par essais successifs.

3. Solution finale pour la construction des triangulations possibles par essais successifs:

a. Algorithme d'essais successifs :

On peut proposer cette stratégie d'essais successifs consistant à prendre à chaque étape de la récursion une des cordes valides et en même temps on maintient à jour un tableau des cordes tracées tout en vérifiant si la corde à ajouter est valide grâce à validecorde.

procédure essais_successifs([tableau des coordonnees] polygone, entier sommets, [tableau de cordes] cordes_tracées, [tableau de cordes] C, [tableau de tableau de cordes] triangulations, réel longueur_actuelle);

Si nombre de cordes_tracées == n - 3 **Alors** // **On a trouvé une triangulation**

Si longueur_actuelle < longueur min **Alors**

longueur min <- longueur_actuelle

On vide le tableau triangulations

On ajoute les cordes_tracées dans triangulations

Sinon Si longueur_actuelle == longueur min

On ajoute les cordes_tracées dans triangulations

retourner longueur_actuelle

Fin Si

Pour chaque corde de C Faire;

Si validecorde(corde[0], corde[1], cordes_tracées) **Alors**

ajouter à longueur_actuelle la distance de la corde

ajouter à cordes_tracées (corde);

essais_successifs(polygone, sommets, cordes_tracées, C, triangulations, longueur_actuelle);

on enlève le dernier élément de cordes_tracées et la distance de la corde de longueur_actuelle; //backtracking

Fin Si

Fait

Fin

b. Complexité de l'algorithme :

La complexité est d'ordre O(n! / (n - 3)!) où n nombre de sommets. Cette complexité résulte du besoin d'explorer systématiquement l'espace des solutions, qui comprend toutes les combinaisons possibles de cordes valides pour construire des triangulations.

L'algorithme fait appel à lui-même de manière récursive pour construire les triangulations, et à chaque étape, il doit tester un certain nombre de cordes pour vérifier si elles sont valides.

Cependant pour des grands polygones la complexité en factorielle peut donner une explosion du nombre d'opérations.

c. Condition d'élagage :

Une condition d'élagage simple peut consister à vérifier si la longueur totale des cordes tracées jusqu'à présent dépasse la longueur minimale trouvée précédemment. Si c'est le cas, on peut arrêter l'exploration de cette branche de l'arbre des appels récursifs.

procédure essais_successifs_elagage([tableau des coordonnees] polygone, entier sommets, [tableau de cordes] cordes_tracées, [tableau de cordes] C, [tableau de tableau de cordes] triangulations, réel longueur_actuelle);

Si nombre de cordes_tracées == n - 3 **Alors** // **On a trouvé une triangulation**

Si longueur_actuelle < longueur min **Alors**

longueur min <- longueur_actuelle

On vide le tableau triangulations

On ajoute les cordes_tracées dans triangulations

Sinon Si longueur_actuelle == longueur min

On ajoute les cordes_tracées dans triangulations

retourner longueur_actuelle

Fin Si

Pour chaque corde de C Faire;

Si validecorde(corde[0], corde[1], cordes_tracées) **Alors**

ajouter à longueur_actuelle la distance de la corde

Si longueur_actuelle < longueur min **Alors**

ajouter à cordes_tracées (corde);

essais_successifs(polygone, sommets, cordes_tracées, C, triangulations, longueur_actuelle);

on enlève le dernier élément de cordes_tracées; //backtracking

Fin Si

On enleve la distance de la corde de longueur_actuelle

Fin Si

Fait

Fin

Cette condition d'élagage nous fait gagner du temps en évitant d'explorer des solutions moins optimales, réduisant ainsi le nombre d'appels récursifs. Les bénéfices se voient surtout sur de grands polygones.

d. Tests sur machine : Solution du plus grand polygone en moins de 2 minutes :

On peut partir jusqu'à 11 points pour avoir une réponse en moins de 2 minutes.

C. Programmation dynamique :

1. Récurrence complète : formule de calcul d'une triangulation minimale

Nous définissons d'abord la fonction de coût suivante :

C(i, j) = coût de la triangulation minimale du sous-polygone si, si+1, ..., sj.

Le cas de base est atteint lorsque j - i <= 2, car il n'y a pas de cordes à tracer pour des sous-polygones avec 2 ou 3 côtés.

C(i, j) = {

C(i, j) = min { distance(si, sk) + distance(sk, sj) + C(i, k) + C(k, j) }, où i + 1 <= k <= j - 1.

C(i, j) = 0 // Si j - i <= 2.

}

La fonction de coût C(i, j) est minimisée en choisissant le meilleur sommet intermédiaire k qui divise le sous-polygone en deux sous-polygones optimaux. Les coûts des deux sous-polygones optimaux sont ajoutés aux coûts des cordes (si, sk) et (sk, sj) pour obtenir le coût total de la triangulation minimale pour le sous-polygone.

2. Algorithme de programmation dynamique

Fonction triangulation_min_dynamique([liste des coordonnees] polygone); **retourne** T, C

On initialise le tableau T a nil

On initialise le tableau C a 0

Pour chaque taille entre 2 et n **Faire**

Pour i allant de 0 à nombres de sommet s- taille **Faire**

j <- i + taille

C[i][j] <- infini

Pour k allant de i + 1 a j **Faire**:

coût <- C[i][k] + C[k][j] + coût de la tentative de triangulation

Si coût < C[i][j] **Alors**:

C[i][j] = cout

T[i][j] = (i, k, j)

Fin Si

Fin Pour

Fin Pour

Fin Pour

Retourne T, C

Fin

3. Complexité spatiale et temporelle de l'algorithme dynamique

La complexité temporelle se calcule à partir des nombres d'appels de récursion qui permettent de remplir la table des couts C.

Σ (j - i - 1) pour i = 0, 1, ..., n-2 et j = i+2, i+3, ..., n-1

Ce qui nous donne une complexité en O(n^3) il y a environ n/2 termes dans chaque somme partielle, et il y a n de ces sommes partielles. Ainsi, la complexité temporelle de l'algorithme en termes de nombre de comparaisons (ou de valeurs de triangulations) est O(n^3).

Comme la matrice des coûts est une matrice carrée n x n alors la complexité spatiale est de O(n^2).

4. Explications, remarques et suggestions pour l'algorithme dynamique

Dans cette stratégie de programmation dynamique, deux sous-problèmes ont toujours un sommet commun, car nous divisons le polygone en deux sous-polygones en choisissant un sommet intermédiaire k. Ainsi, les deux sous-polygones partagent les sommets i, k et j. Cette façon de décomposer le problème est intéressante du point de vue des triangulations considérées, car elle permet de diviser le problème en sous-problèmes plus petits et indépendants, tout en tenant compte des interactions entre les sous-polygones adjacents. Cela permet de résoudre le problème de manière optimale en exploitant la structure du problème et en évitant les redondances.

Si la décomposition se faisait en tirant deux cordes quelconques, l'algorithme devrait être modifié pour tenir compte de la manière dont les deux cordes interagissent et se coupent éventuellement. Cela compliquerait l'algorithme, car il faudrait s'assurer que les deux cordes ne se coupent pas et que les sous-polygones formés sont toujours convexes. De plus, cela augmenterait la complexité de l'algorithme, car il faudrait examiner un plus grand nombre de combinaisons possibles de cordes pour résoudre le problème.

D. Algorithme glouton

1. Implémentation de l'algorithme glouton

Function glouton_tour_des_sommets(liste_de_coordonnés_polygone) retourne triangulation:

Début:

n←nombre_de_sommet(liste_de_coordonnés_polygone)

sommet_courant ← 0

cordes← liste_vide

Tant que nombre(cordes) < n - 3 **faire**:

Pour j allant de 1 à n **faire**:

sommet_suivant ← (sommet_courant + j) % n

Si la corde (sommet_courant, sommet_suivant) est valide:

cordes.append((sommet_courant, sommet_suivant))

sommet_courant = sommet_suivant

sortir du pour.

Fin si

Fin pour

Fin tant que

retourne cordes

FIN

La fonction `glouton_tour_des_sommets` effectue une triangulation gloutonne d'un polygone en parcourant les sommets de manière circulaire. Voici un résumé de ce que fait le code :

1. Il détermine le nombre de sommets `n` du polygone.
2. Il génère les côtés du polygone (les arêtes reliant les sommets consécutifs) et les stocke dans la liste `cordes_tracees`.
3. Il crée une liste vide `cordes_internes` pour stocker les cordes internes de la triangulation.
4. Il initialise le sommet courant à 0 (premier sommet du polygone).
5. Tant que la liste des cordes internes n'a pas atteint la taille `n-3` (un polygone à n sommets doit être divisé en n-3 cordes internes) :
    a. Parcourir les sommets du polygone dans l'ordre circulaire à partir du sommet courant.
    b. Pour chaque sommet suivant, vérifier si la corde reliant le sommet courant et le sommet suivant est valide (ne croise pas les cordes déjà tracées) en

utilisant la fonction `validecorde`.
   c. Si la corde est valide, l'ajouter aux listes `cordes_internes` et `cordes_tracees`.
   d. Mettre à jour le sommet courant pour qu'il soit égal au sommet suivant et continuer la boucle.
6. Retourner la liste des cordes internes de la triangulation gloutonne.

2. Remarques a propos de l'exactitude de la stratégie gloutonne

Bien que la stratégie gloutonne soit simple à mettre en œuvre et puisse donner des résultats satisfaisants dans certains cas, elle ne garantit pas une solution optimale pour la triangulation d'un polygone convexe en général. Pour obtenir une solution garantie optimale, il est préférable d'utiliser des méthodes plus sophistiquées, telles que la programmation dynamique présentée précédemment.

E. Recommandation argumentée

Après avoir étudié les différentes méthodes pour résoudre le problème de la triangulation d'un polygone convexe, voici une recommandation argumentée :

1. Essais successifs : Cette méthode explore toutes les possibilités de triangulation, ce qui entraîne une complexité élevée. Malgré l'utilisation d'élagage pour réduire le nombre d'appels récursifs, cette méthode n'est pas efficace pour les polygones ayant un grand nombre de sommets.

2. Programmation dynamique : Cette méthode est plus efficace que les essais successifs, car elle évite la redondance des calculs en mémorisant les résultats intermédiaires. La complexité temporelle de cette méthode est de l'ordre O(n^3), et sa complexité spatiale est de l'ordre O(n^2). Cela rend cette méthode plus adaptée pour traiter des polygones de taille plus importante.

3. Algorithme glouton : Bien que cette méthode soit simple à mettre en œuvre et rapide, elle ne garantit pas une solution optimale pour tous les polygones convexes. Elle peut donner des résultats satisfaisants dans certains cas, mais il n'est pas recommandé de l'utiliser si l'optimalité de la solution est cruciale.

Recommandation : La méthode de programmation dynamique est recommandée pour résoudre le problème de la triangulation d'un polygone convexe. Elle offre un bon

compromis entre la complexité et la garantie d'une solution optimale. Cette méthode est préférable aux essais successifs, qui ont une complexité plus élevée et ne sont pas adaptés aux polygones de grande taille, ainsi qu'à l'algorithme glouton, qui ne garantit pas l'optimalité de la solution.